Personal Computer : MSX　MSX2　MSX2+ MSX turboR

CPU : Z−80A

DOS : MSX−DOS or MSX−DOS2

Assembler : Coral Simple ASM Version 2.0

MSXで学ぶZ−80A　アセンブラ入門

Z−80A　Assembler

Personal Computer : MSX MSX2 MSX2+ MSX turboR

CPU : Z−80A

DOS : MSX−DOS or MSX−DOS2

Assembler : Coral Simple ASM Version 2.0

*MSX*で学ぶ*Z−80* アセンブラ入門

Z−80 Assembler

はじめに

パソコンを使って皆さんは何をしていますか？

ゲームで遊んでいる人、ワープロとして使っている人、**BASIC** 言語を使ってプログラムを作っている人。　　. . . ひとつのパソコンでいろいろな楽しみ方ができます。

そこで、今までのパソコンの使い方以外の楽しみ方をここで提案します。それは、アセンブラ言語を使ってのマシン語プログラムの開発です。

アセンブラ言語を学習する事により、パソコンの仕組みが今まで以上に理解することができるようになります。

アセンブラ言語でプログラムを作成するには、アセンブラと呼ばれるプログラムが必要になります。**MSX** を使っている方は幸いにして、当社から *Simple ASM* という **MSX** 用のアセンブラを購入することができます。

本書では *Simple ASM (Ver2.0)* を使ったアセンブラの入門書に仕立てあげました。初めてアセンブラ言語勉強する人や、今ひとつアセンブラのプログラムの作り方がわからなかった方にとって最適な参考書です。今まで何回もアセンブラのプログラムを作ったことがある方には物足りないかも知れませんが. . .

本書は、短いプログラムをいくつか掲載していますが、実際に入力して実行して見て下さい。パソコン学習、特にアセンブラ言語の学習はより多くのプログラムを実行することが理解の早道になります。

Z−80の機械語を一日も早くマスターし、楽しくおもしろいプログラム作って、今までとはちょっと違うパソコンライフを楽しんで下さい。

1993. 5

■注意

・本書と実際の結果に差異があった場合には、実際の結果が優先します。
・本書の内容については万全を期して執筆していますが、万一ご不審な点や誤り、記載もれなどがある場合はご容赦願います。
・実際の操作においての運用結果については一切の責任を負いかねますのでご了承ください。
・本書の内容の一部または全部を無断転載することは固く禁止しています。

目　次

第1章　マシン語の基礎

第2章　アセンブラの基礎

第3章　Z-80の基礎

第4章　Simple ASMの基礎

第5章　Simple ASM操作法

第6章 Z-80命令のすべて

第7章 疑似命令

第8章 ファンクションコール

第9章 ＢＩＯＳ

Appendix

第１章
マシン語の基礎

本章ではアセンブラの勉強を始める前に、最低限知っておかなければいけないコンピュータの基礎知識をまとめてみました。この章の内容を理解することがアセンブラ学習の第一歩になることでしょう。

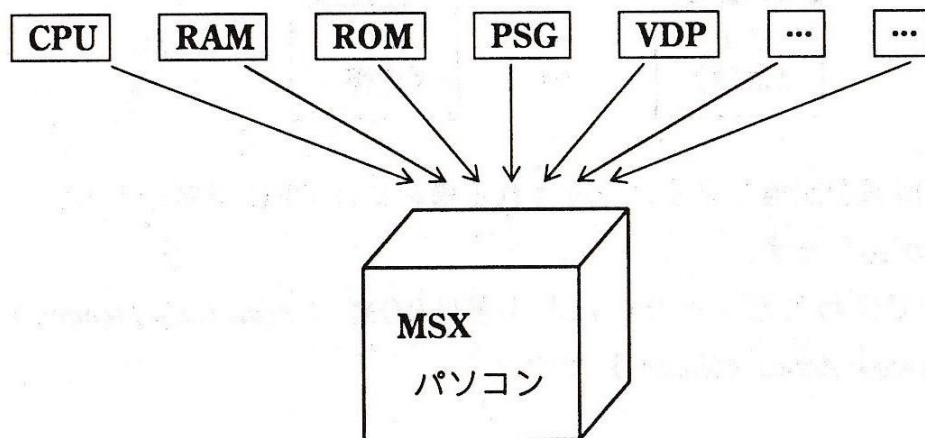
1−1 コンピュータの構成

私たちはコンピュータショップで「このパソコンの **CPU** は **R**800で…」、「**BASIC** は **ROM** に…」、「**RAM** の容量は…」などという会話をよく耳にします。また、パソコンのカタログや、マニュアルの仕様にも、**CPU** が何、**ROM** の容量がいくつとかが書かれています。

ここでは簡単にパソコンの構成を学習します。

まず、**CPU** はパソコンの中心部分です。**MSX** では **Z**−80や **R**800が使用されています。こんなことは常識ですね。

そして、メモリとなる **RAM** と **ROM** があります。それ以外にも、**PSG**、**VDP** などの集積回路（**IC**、**LSI** など）もあります。

最初に、**CPU**、メモリ（**ROM**、**RAM**）について解説しましょう。

CPU（*Central Processing Unit*）

CPU（中央演算処理装置）は、その名の通りマイコンシステムの頭脳にあたる部分です。ほとんどの命令はここで解読され、実行されます。この **CPU**

が実行できる命令には加減算や判断、外部装置の制御を行うものなどがあります。

CPU は各 **IC** メーカーから数多くのものが発表されていますが、取り扱うデータの規模に応じて８ビット、16ビット、32ビットなどの種類があります。
既にご存知のように **MSX** には８ビット **CPU** の **Z−80A** が搭載されています。
機種によっては、 **R**800という **CPU** も使われています。

R800は **Z−**80の命令を全てカバーし、さらにいくつかの命令が追加された優れものです。

メモリ（ROM/RAM）

メモリとは記憶素子のことを言います。

コンピューターシステム内の記憶素子には、ＲＯＭ(*Read Only Memory*)と **RAM**(*Random Access Memory*) とがあります。

コンピュータは基本的に **CPU** とメモリ間でデータのやりとりを行います。
例えば 10＋20 の計算（幅広い意味で演算といいます。）を行うときには、メモリ上の10、20というデータを **CPU** に送り、 **CPU** はそこで計算し、その結果をメモリに戻します。

メモリは情報を記憶します。記憶される情報には **CPU** が実行するプログラムとデータがあります。

パソコンで使われるメモリのほとんどは**ROM**（*Read Only Memory*）と **RAM**（*Random Access Memory*）です。

ROMとは

読み出し専用のメモリで、通常データを書き込むことはできません。そのかわり電源を切ってもデータは消えません。 **CPU** を起動させるために必要です。

BASIC インタプリタはここに記録されているため、電源を入れるだけで基本システムが起動するわけです。

RAM とは

読み出し、書き込みができるメモリです。しかし電源を切るとデータは消えてしまいます。ユーザの作成したプログラムやデータはここに格納します。また、プログラムの誤動作などによりプログラム自身を破壊してしまう可能性があるので、重要なプログラムは実行前にフロッピィディスクにセーブしておく習慣をつけましょう。

入出力Ｉ／Ｏ（*Input/Output*）

ポート（*Port*: 港）などとも呼ばれ、主に周辺装置とのデータのやりとりに使われます。入出力ポートはメモリと同じように１バイトのデータを取り扱うので、メモリのように考えても差し支えないでしょう。また **Z–80**ではメモリ空間と **I/O** 空間とが独立しているのでメモリ全てをプログラムやデータのために使うことができます。

1–2 アドレス

人間の家は「住所」によってその所在地が明らかになりますが、コンピューターの世界でも「データ」の所在地を「アドレス (*Address*: 住所)」で参照します。しかし数字に強い **CPU** は「データ」の居所を「数値」だけで参照します。このためアドレスは「番地」と訳されます。

アドレスはメモリに付けられています。いちばん番号の若いアドレスは0番地で、その次は1番地、2番地と言うように段々と大きくなっていきます。人間界の何丁目何番地を1つの数値で表したと思えば分かりやすいでしょう。 **CPU** はメモリをアドレスの小さい順（昇順）に読んでいきます。リセット直後の

CPUはまず0番地のデータを参照して、仕事を開始します。

ひとつのアドレスには1バイト（8ビット）のデータがあります。

Z−80は0番地から65535番地までを管理することができます。

0番地	←それぞれ1バイトのデータが格納されている。（メモリには、プログラムとデータの区別はなく、全て単なる数値に過ぎない。）
1番地	
2番地	
⋮	

1−3 バスとは

CPUとメモリやI/Oを接続しているラインのことをバス（*Bus*）と呼びます。そのバスの上には各種の電気信号が流れています。

バスの詳細な管理は全てCPUがやってくれるので、ここでは紹介するにとどめます。Z−80のバスには3種類あります。

アドレスバス

16本のピンで構成され、メモリ、I/Oのアドレスを選択します。

データバス

データの通り道です。8本のピンで構成されています。CPUからメモリへ、そして、メモリからCPUへデータがやりとりされるときなどに使われます。

コントロールバス

CPUからの指令の通り道で、周辺のICを直接コントロールします。

1−4 数の表記

これまでに「バイト」や「ビット」という言葉が現れてきましたが、これらを含めて、コンピューターで使う「数」について説明します。

２進数

コンピュータを勉強していく上で、よく2進数、16進数という言葉を使います。まず最初に、なぜ、コンピュータの世界で２進数基本なのかを説明しましょう。

コンピュータの情報の最少単位はビット（*bit*）と呼ぶことから始まります。ビットは“0”と“1”を使って表します。コンピュータの電気的信号では「ある」、「ない」を簡単にしたものです。「ある」は電圧があるということで、コンピュータの世界では“1”で表し、「ない」は電圧がないということで、“0”で表します。

１つの情報は“0”と“1”で表しますが、情報が２つあった場合は、00、01、10、11の４種類があります。情報が２つあるというのはつまり２ビットです。

では情報が８個あった場合、つまり８ビットの時の“0”と“1”の組み合わせを考えてみましょう。

```
0000 0000
0000 0001
0000 0010
0000 0011
0000 0100
0000 0101
    |
    |
    |
1111 1110
1111 1111
```

結果は256種類あります。つまり2^8になるわけです。なぜここで８ビットの

組み合わせを考えたかというと、８ビットというのは**Z−80**で取り扱うことのできる１つのデータの単位だからです。

２進数を10進数に変換

では２進数を私たちが簡単に理解できる10進数に変換してみましょう。そのためには、２進数の各桁に2^nを掛けて合計します。

例えば、１００１００１１の時は次のようになります。

つまり右のビットから順に1、2、4、8、16、32、64、128の意味を持つことになります。

“1”と“0”の８桁の組み合わせ256種類を０から255に割り当てられるのです。

MSX−BASICでは２進数の頭に＆Ｂをつけて表しますので、次のように打ち込めば簡単に10進変換ができます。

さきほどの147を確かめるには、

```
PRINT &B10010011
```

とします。

10数を２進数に変換

では次に10進数を２進数に直してみましょう。

さきほどは１、２、４、８、16、32、64、128で表されているところの足し算をしましたが、今度は引き算をします。

例えば147は、まず147から128（＝2^7）を引きます。引くことができれば2^7の位置に“1”を、引けなければ“0”を書きます。ここでは“1”になります。

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1							

次に引いた残りから64（＝2^7）を引きます。ここでは19から64は引けません。同じように 32、16、8、4、2、1と繰り返し引き算を行います。

$147-2^7$	＝	19	引ける	→	1	←―― 最上位ビット
$19-2^6$	＝	×	引けない	→	0	
$19-2^5$	＝	×	引けない	→	0	
$19-2^4$	＝	3	引ける	→	1	
$3-2^3$	＝	×	引けない	→	0	
$3-2^2$	＝	×	引けない	→	0	
$3-2^1$	＝	1	引ける	→	1	
$1-2^0$	＝	0	引ける	→	1	←―― 最下位ビット

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	0	0	1	1

となります。

MSX−BASIC では次のようにします。

```
PRINT BIN$(147)
```

で簡単に２進数に変換できます。

２進数の足し算

２進数も数字ですから当然四則演算が行えます。ここではまず足し算を考えてみましょう。１桁の２進数は“0”と“1”ですから、足し算は次の４種類になります。

```
0 + 0 ➡  0
0 + 1 ➡  1
1 + 0 ➡  1
1 + 1 ➡ 10
```

最後の１＋１＝１０というのだけが10進数のときと答えが異なります。10進数では“2”ですが、２進数では“0”と“1”だけですから、隣の桁に繰り上がります。この桁をキャリーといいます。

```
    1
 +  1
 ----
   10
   ↑
   キャリー（下からの桁上がり）
```

次に５＋７を考えてみましょう。
それぞれ 00000101 と 00000111 になります。

```
     00000101
 +   00000111
 ------------
     00001100
          ↑↑
   この桁は下位からの桁上がりも加える
```

```
    1          0
    1          1
 +  1       +  1←桁上がり
 ----       ----
   11         10
```

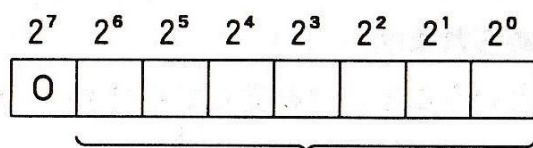
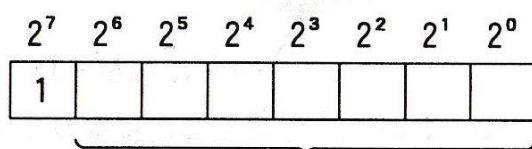
負の数の表し方

今までは正の数だけを考えてきましたが、数字ですから当然負の数も存在します。負の数は２の補数表現という方法で表されます。

これは最上位のビット（桁）が１の時に負の数として扱います。８ビットの時は第７ビット（最下位ビットは第０ビット）が符号ビットとなります。８ビ

ットのうちの１ビットを符号に取られてしまったので正の数、負の数それぞれ表せる範囲は2^7＝128通りづつとなりました。

正の数

負の数

“０”と“１”の８桁の組み合わせは256種類ですが、正の数は00000001を＋１、00000010を＋２としていけば次のようになります。

```
0000 0001      1
0000 0010      2
0000 0011      3
0000 0100      4
   |           |
0111 1111    127
```

０は全く変わらず00000000です。負数は、11111111を－１とし、11111110を－２という形で割りふります。

```
1111 1111     −1
1111 1110     −2
   |           |
1000 0000   −128
```

つまり一番上位のビットを負の記号としてみれば、８ビットで表せる正数は

−128〜127になります。

負の数の求め方

この補数という方式で作られた負の数は、正の数のすべてビットを反転したものに＋１することにより簡単に求められます。

ここでは、このような関係があるということだけを覚えてください。理論は専門書にまかせましょう。

たとえば−15をつくるときは＋15を２進数に変換してから行います。

＋15を２進数に	00001111
すべてのビットを反転させる	11110000
＋１をする	11110001

← これが−15

16ビットで数値を表すときも最上位ビットが正負を表すことになります。この最上位ビットのことをサインビットと呼びます。

MSX−BASICで

```
PRINT &B1000000000000000
```

としてみてください。

MEMO

●２進数８ビットで表現できる数は

最上位ビットをサインビットとしない場合	0〜 255
最上位ビットをサインビットとした場合	-128〜 127

●２進数16ビットで表現できる数は

最上位ビットをサインビットとしない場合	0〜65535
最上位ビットをサインビットとした場合	-32768〜32767

16進数

CPUが直接取り扱うデータは２進数ですが、“０”と“１”の組み合わせ

は人間にとって非常にまぎらわしい性格をしていますので、コンピュータを使う人は16進数をよく使います、２進数４桁が16進数の１桁に対応するので非常に便利です。

16進数は１桁で10進数の０～15までを表すため、10進数の10～15は16進数でＡ～Ｆのアルファベット１文字に対応させています。

10進数 、２進数、１６進数対応表

10進数	２進数	16進数
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

ここで10進、２進、16進数の対応を作ってみました。特に２進数は４桁で16進数の１桁になることを頭に入れ、完全に暗記してください、

２進数から16進数に変換

２進数を16進数に変換する方法をここではマスターしましょう。先ほどの対応表を暗記していればとても簡単です。

手順としては、まず16ビットの２進数を下位の桁から４桁ごとに区切ります。この例では16ビットですので上位から区切っても結果は同じですが、下位の桁から区切るように覚えてください。次に区切られた４ビットごとに16進数に変

換してください。変換方法は暗記した２進、１６進数変換表をもとにします。これで終わりです。

例“1100110101011101”を16進数にする。

①下から４桁ごとに区切る。

1100 1101 0101 1101

②それぞれ４ビットを１桁の16進数に変換する。

1100 1101 0101 1101
↓ ↓ ↓ ↓
C D 5 D

16進数から２進数に変換

では次に16進数を２進数に変換する方法をマスターしましょう。先ほどの２進数からの16進数に変換した方法がわかれば簡単です。

16進数は０～９、及びＡ～Ｆで表されますから、それぞれの数値を４桁の２進数にしていきます。

例“７Ａ６５”を２進数にする。

①１桁ごと区切る。

７ Ａ ６ ５

②それぞれを２進数に変換する。

７ Ａ ６ ５
↓ ↓ ↓ ↓
0111 1010 0110 0101

MSX−BASICを使った変換方法

では、MSX−BASICを使って、２進数、16進数、そして、変換方法をみてみましょう。

２進数は **MSX−BASIC** では **&B** を頭につけます。同じように16進数は **&H** をつけます。

たとえば、２進数の1101は&B1101になり、16進数の8FEDは&H8FEDというように表します。

MSX-BASICで次の命令を実行してみましょう。

```
PRINT &B11010101
PRINT &B0111
PRINT &HFE3
PRINT &H789E
PRINT &B1100 + &HEF
```

次に、ある数値を16進数で表す方法を説明します。ここでいうところのある数値とは２進数でも10進数でも構いません。

16進数に変換する方法は HEX$() という関数を使います。結果は文字になります。

２進数に変換する方法は BIN$() という関数を使います。結果は HEX$() のときと同じように文字になります。

MSX-BASICで次の命令を実行してみましょう。

```
① PRINT HEX$(&B11010101)
② PRINT HEX$(9600)
③ PRINT BIN$(&H1234)
④ PRINT BIN$(1234)
```

上の例題で、①は２進数の“11010101”を16進数で表示します。
同じように②は10進数の“9600”を16進数で表示します。

③は16進数を２進数で表示し、④は10進数を２進数で表示させています。
実際に **MSX** で実行してください。

数値表現の決まり

"1010" という数字が仮にあったとしたときに、これは２進数なのか16進数なのか、10進数なのかわかりません。ですから、ここで数値の区別をするため２進数の最後にはＢという文字を、16進数の最後にはＨという文字を付加します。Ｂは*Binary*、Ｈは*Hexadecimal*の頭文字をとったものです。

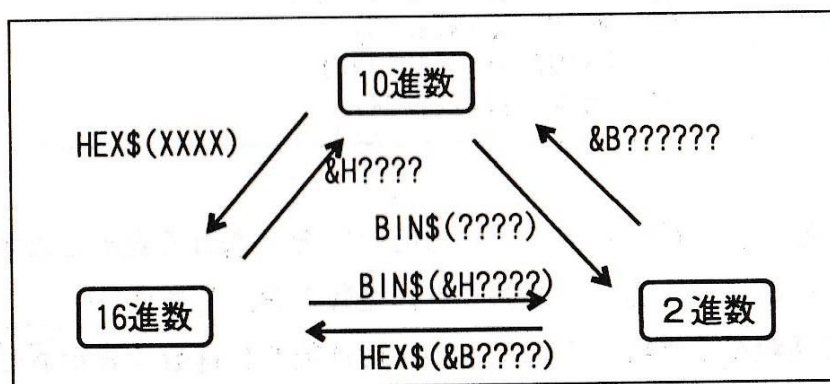
例	14	…………	10進数の14です。
	14H	…………	16進数です。10進数にすると20になります。
	1110B	…………	２進数です。10進数の14です。

また、16進数の場合は文字列と区別するために、頭の１桁がＡ～Ｆで始まるときは０を付加することがあります。特にアセンブラプログラム中で記述するときはには、必ずつけなければいけません。

例	E755H	→	0E755H
	B5H	→	0B75H

表記の上で明らかに２進数、16進数とわかっているときには、Ｂ、Ｈを省略するときがあります。

まとめ

２進数、16進数、10進数の変換を**MSX－BASIC**を使って変換する関係は次のようになります。

文章中では、　２進数のとき最後に "Ｂ" をつける
　　　　　　　16進数のとき最後に "Ｈ" をつける
　　　　　　　10進数のときは何も付加しない。

第２章
アセンブラの基礎

本章では、アセンブラのプログラムを作るための基本的な用語、そして、プログラムを作成するための流れを説明します。

2-1 プログラムとは

プログラムはコンピュータが理解し実行する命令の集まりですが、BASICで書かれたプログラムとZ−80自身が理解する機械語（マシン語）とでは、同じ処理でも大きく違ったプログラムとなります。

Z−80機械語プログラムの方は２進数で書かれています。これをマシンコードと言うこともあります。この機械語は、ふつうの人が見ても全く解りません。

この機械語を一連の流れに従って順序正しく組み合わせてできたものが機械語プログラムです。順序正しい流れをしていないプログラムは、プログラムになり得ず、与えられた仕事を確実に行うことはできません。

コンピュータの命令を規則正しく並べていくのがプログラマであるあなたの仕事であり、腕の見せどころとなります。

機械語のプログラムを作成するためには、人間に優しくわかりやすくしたものをアセンブリ言語とよびます。これからじっくり機械語とアセンブリ言語について解説しましょう。

2−2 機械語とアセンブリ言語

機械語

まず機械語から説明しましょう。

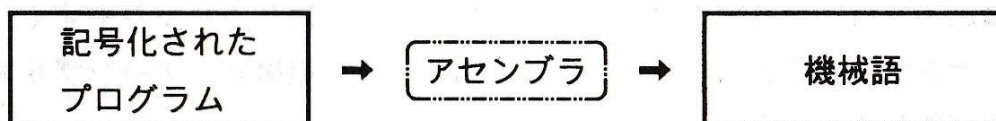
機械語は一見単なる数値の羅列にしか見えません。人間にとってはプログラムかデータかの区別さえつかないものです。通常、機械語は16進数で表記されますが、基本的には２進数です。ただ、16進数を使った方が人間にとって非常

に便利なためよく16進数を使用します。例えば、10＋８の計算をするプログラムは次の通りです。

（例）10＋８のを行う機械語プログラム

16進表現		2進表現
	３Ｅ	0011 1100
	０Ａ	0000 1010
	Ｃ６	1100 0110
	０８	0000 1000

アセンブラ

機械語とはどんなものかを説明しました。**CPU** は機械語の基づいて処理をしていますが、私達には１６進数の羅列を見ても全く意味不明です。そこで、人間にとって分かりやすいようにするため、機械語を記号に置き換えてプログラムできるようにします。この記号化されたプログラム言語がアセンブリ言語です。そして、この記号化されプログラムを機械語に変換する道具（ソフトウェア）をアセンブラ（*Assembler*）と呼びます。

ニーモニック

機械語は数値でしたがここでは人間の解る記号に置き換えます。この記号のことをニーモニック（*Mnemonic*）と言います。ニーモニックは疑似命令と呼ばれるものを除いて、機械語と１対１で対応しています。10＋８の計算は16進機械語で、3E 0A C6 08でしたがニーモニックを使うと、次のようになります。

10＋８をニーモニックであらわす。

```
LD    A,10
ADD   A,8
```

このLD（Load）,ADD（Add）の記号のことをニーモニックと呼び、それぞれ「データをロード（移動）する」、「足す」の意味です。このようにニーモニックのほとんどは英語の省略形なので使っているうちにすぐに慣れるでしょう。

アセンブリ言語

ニーモニックを使って表される記号言語をアセンブリ言語と呼びます。

また、アセンブリ言語で書かれた一連のプログラムをソースプログラムと呼びます。

ソースプログラムとオブジェクトプログラム

機械語のプログラムはまず、ニーモニックを使ったアセンブリ言語で記述されます。このプログラムのことをソースプログラムと呼び、これを機械語に「翻訳」したものをオブジェクトプログラムといいます。では先ほどの10＋８の例をもとにソースプログラムとオブジェクトプログラムの関係をみてみましょう。

ソースプログラム

```
LD   A,10
ADD  A,8
```

➡

オブジェクトプログラム

```
3E 0A
C6 08
```

2−3 プログラムを作るプログラム

ここで言う「プログラムを作る」とは、コンピュータが理解できる機械語プログラムを作成することにほかなりません。つまり、先ほどの「オブジェクトプログラム」を作ることを意味します。オブジェクトプログラムは当然のことながら、ソースプログラムを「翻訳」して作成されます。

アセンブラソースをオブジェクトに変換するためのプログラムが「アセンブラ」です。さっきのような短いプログラムであれば、ニーモニック一覧表を見れば簡単にマシンコードに翻訳することができます。このように人間の手でアセンブルする事を「ハンドアセンブル」と言います。アセンブルとはソースプログラムからオブジェクトプログラムを生成する翻訳作業のことです。

2−4 アセンブリ言語プログラムの基礎

アセンブラはアセンブリ言語で書かれたソースを忠実に機械語に変換するプログラムです。しかし、アセンブラがソースプログラムを自動的に生成することはありません。あなた自身がソースプログラムを作成し、それをアセンブラに投入することによって、初めてオブジェクトができあがるのです。それではここで、プログラムを作る上での基礎知識について触れてみましょう。

プログラムを作る上で考えなければならないのは、どんな目的でどのような処理をしなければいけないか、ということです。プログラム完成までの大まかな手順を以下に示します。

1．目的の決定
2．処理概要の設計
3．流れ図
4．コーディング
5．プログラムテスト
6．完成

目的の決定

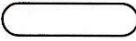
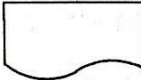
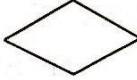
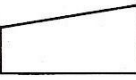
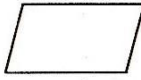
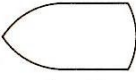
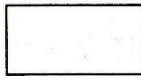
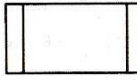
みなさんがどういう目的でMSXを使っているかをたずねたとき、その答えは各々ばらばらだと思います。ゲームを作りたいと思っている人、勉強をしたいと思っている人。その人たちにはプログラムというものが必要になってきます。

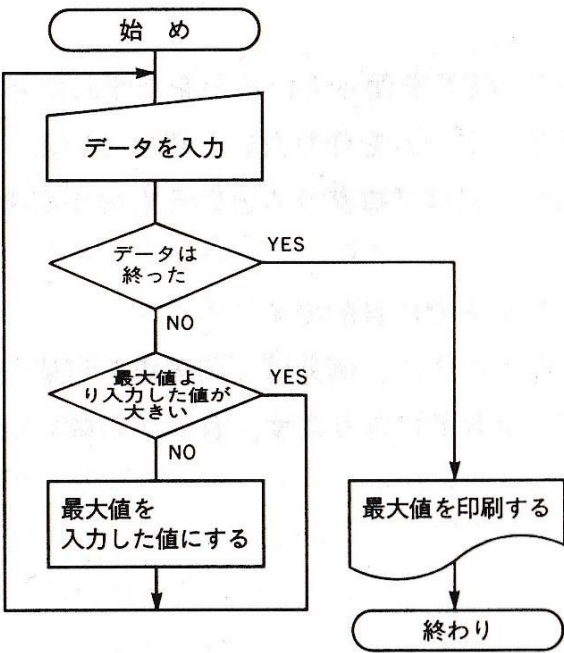
ゲームを作りたい人はそれがすでに目的です。

勉強をしたいと思っている人の中で、例えば「円周率を可能な限り求めたい」と考えたならば、これも立派な目的になります。目的が明確にならない限り、先には進めません。

処理概要の設計

目的が定まったならば、次はこれを達成するために、例えばゲームならばキャラクタを考えたり、ストーリーを練ったりしなければなりません。このように、キャラクタやストーリーを決めるというのも処理概要の設計です。難しく考える必要は全くありません。

流れ図（フローチャート）

流れ図とは処理の流れを図形で表したものです。流れ図には様々な種類のものが発表されていて、使用する言語（BASICとか機械語とか）によって使い分けられます。プログラムの理解を図形に託したもので、非常に見やすく、考え易くなります。本書ではJISで規格が定められている「フローチャート」を紹介します。

処理の概略をつかむためには「フローチャート」が良いでしょう。ちなみに、「キーボードから入力した値の中から最大値を求めてそれを印刷する」という処理を表すフローチャートは、次のようになります。

流れ図に使う記号はＪＩＳ規格で決められています。また、流れ図の記号を書くのに便利な便利なテンプレートが文房具店やパソコンショップで買うことができ、そのテンプレートの説明書にも使い方が記載されています。プログラムを作る人は是非テンプレートを用意したいものです。

記号	説明	記号	説明	記号	説明
端　子	開始・終了などに使います	入出力	プリンタ等の印字を表わします	判　断	条件判断等に使います
手入力	キーボード等の入力処理に使います	入出力	一般的な入出力を表わします	結合子	流れ図の線が複雑になったり，他のページにまたがるときなどに使われます
表　示	画面表示に使います	処　理	計算などの処理に使います	サブルーチン処理	まとまった処理ルーチンを使うときに使います

コーディング

コーディングとはコーディングシートなどにプログラムを書いていくことです。コーディングシートとはプログラム記述用の特別な用紙です。短いBASIC プログラムなどであればキーボードを直接たたいて、考えながら入力しますが、アセンブリ言語では紙と鉛筆を使って流れに従ってコーディング（記述）していった方が無難です。流れ図ほどではないにせよ、プログラム全体が見渡せるわけですからミスの発見にも役立ちます。たった1語の命令の誤りから、マシンがウンともスンとも言わなくなってしまう状態に陥ること（プログラムの暴走）が幾分防げます。本書の巻末にはハンドアセンブルの場合も考慮したコーディングシートを掲載しています。コピーを取ってお使い下さい。

プログラムテスト（デバッグ）

プログラムテストとは、コーディングシートに書かれたプログラムをハンドアセンブルしたり、アセンブラを使って機械語に直し、コンピュータを実際に使ってプログラムが目的の処理を行うかどうかをテストする過程のことです。もし期待に反して間違った結果が出てきた場合にはプログラムにミスがある場合、考え方自体に無理がある場合などがあげられます。こういう状態のことを、プログラムにバグ（*Bug:* 虫）がいるといい、これを修正することをデバッグ（*Debug:* 虫取り）と言います。最初に作成したソースプログラムが1発で成功することは、むしろ希で、デバッグを繰り返すことにより1本のプログラムが完成します。

完成

機械語プログラムが完成したら、今度はこれをフロッピィディスクに記録します。もしくは MSX マシンの特徴でもあるＲＯＭカートリッジに焼き込むことも考えられます。

また、BASICプログラム中に機械語プログラムを挿入するときには、BASIC の DATA 文に書き換えるなどの加工をします。

さらに、MSX−DOS 用に開発したプログラムであれば DOS の外部コマン

ドとして実行できます。

2−5 その他の基礎知識

エディタプログラム

ソースプログラムを人間が開発したなら、それをコンピュータに入力する作業が必ず必要になってきます。この入力・編集作業を行うのがエディタです。またエディタには、プリントアウトができたり、指定文字列の検索、置換などが行えるものもあります。

（編集機能）

- ・入力　　データを入力する。
- ・修正　　データを修正する。
- ・リスト　プログラムのリストを取る。

ファイル

ソースプログラム、オブジェクトプログラムをそれぞれフロッピィディスクに記録したものをファイルといいます。ソースファイル、オブジェクトファイルなどといって、ファイルの内容を明確に分けて呼びます。

ファイル操作

エディタプログラムはソースプログラムをエディト（編集）する以外に、次のようなファイル操作が可能です。

・ソースプログラムファイルの入力（ロード）…ファイルからメモリへ
・ソースプログラムファイルの出力（セーブ）…メモリからファイルへ

モニタ

モニタはメモリの中身を見るためのプログラムです。一般にはメモリに直接データを書き込んだり、データを画面に表示する機能は必ず持っています。さらに、デバッガと呼ばれる強力なモニタになるとCPU内のレジスタ表示／変更、トレース（オブジェクトを１命令づつ実行させながら、その時のレジスタの内容を表示する機能）やブレイクポイント（指定したアドレスでプログラムを終了すること）を設定できるものなど様々です。また機種によっては（貧弱ではあるが）モニタ機能を初めから搭載しているものもあります。

第3章

Z-80の基礎

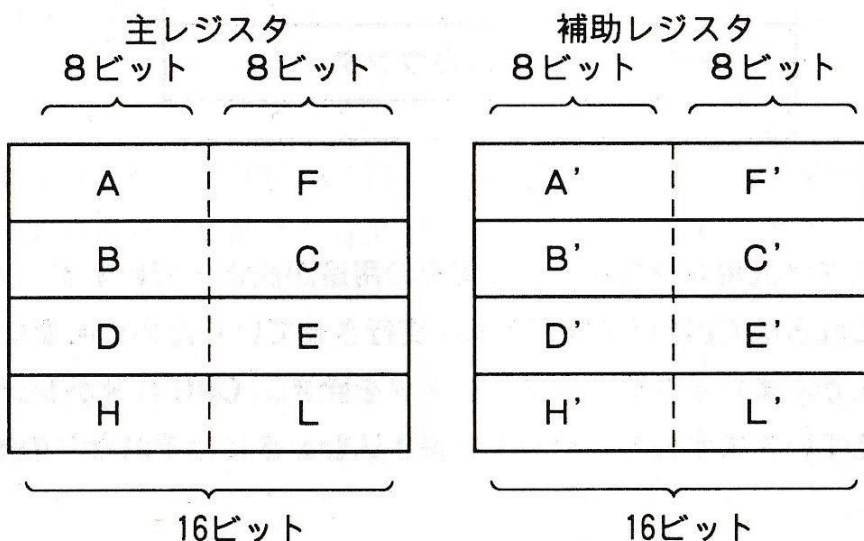
本書はZ−80のアセンブラのプログラムを作成するのが目的です。そのためにはZ−80というCPUがどんなCPUなのかを理解する必要があります。本章ではこのZ−80についての基礎的な知識をまとめました。

3−1 レジスタ

レジスタの種類

Z−80はレジスタと呼ばれるメモリ領域を持っていますが、これはＲＯＭやＲＡＭとは性質が異なり、CPUが演算をするためにデータを一時的に記憶する働きがあります。

Z−80のCPUには22個の独立したレジスタがありますが、そのうちの８個は補助レジスタと呼ばれ、主レジスタと同時に使うことはできません。またフラグもレジスタの一部ですが特別な意味を持ちます。以下にこれらを紹介します。

汎用レジスタ

ここまでの各レジスタはとなり同士をペアにして16ビットレジスタにすることができます。例えば、Ｂレジスタを上位８ビット、Ｃレジスタを下位８ビッ

トとした16ビットレジスタはＢＣペアレジスタとなります。上下の逆転や、となり同士でない組み合わせはできません（例：ＥＤレジスタ、ＡＣレジスタなど）。　以上は「汎用レジスタ」ですから、プログラム上、必要に応じて自由にデータを読み書きすることができます。各種の演算やデータ転送にはこれらのレジスタを使用しますのでよく理解しておいて下さい。また補助レジスタの機能は主レジスタと同等です。

主レジスタのことを表レジスタ、補助レジスタのことを裏レジスタと呼ぶこともあります。

専用レジスタ

専用レジスタは汎用レジスタと違ってその用途が決まっています。大ざっぱにみると、これらは**CPU**がプログラムを実行させていくために必要なものといえるでしょう。またインデックスレジスタを除き、**CPU**自身がレジスタの値を書き換えていきますので、データを書き込むときにはそれなりの知識を必要とします。

各レジスタの特徴

レジスタにはそれぞれ特徴があります。次に各レジスタの特徴を示します。
ここでの説明には突然難しい言葉がでてきますが、実際にプログラムを組む段階になれば理解できるようになります。ここでは、気軽に一読してください。

Aレジスタ（アキュムレータ）

８ビット処理の中心となるレジスタで各種演算の主役です。Ａレジスタに関する命令が最も多いことからも納得できるでしょう。

Fレジスタ（フラグレジスタ）

このレジスタは特殊なので別に説明します。

Bレジスタ

Ｃレジスタとペアで16ビットＢＣレジスタとして使うことができます。
単体では**DJNZ** 命令のループカウンタに使わます。

Cレジスタ

入出力命令でポートの指定に使われます。ＢＣレジスタとして16ビットループカウンタしてつかってもよいでしょう。

Dレジスタ

Ｅレジスタとペアで16ビットＤＥレジスタとして使うことができます。
ブロック転送命令の目的アドレスポインタとしても使われます。

Eレジスタ

単体ではほとんど使われないためレジスタの退避に使う事があります。
しかしＤレジスタとペアになることが多い、ポインタとしても使われます。

Hレジスタ

Ｌレジスタとペアにすることが多く16ビットデータの処理に適しています。

またアドレスポインタとして使われることもありHレジスタ単体での使用はむしろ希です。16ビット命令はHLレジタに関するものが最多です。

Lレジスタ

Hレジスタと同等の機能を持ってます。

Iレジスタ（割り込みベクタ）

割り込みベクタテーブルのアドレスの上位 1バイトを記録しています。あらかじめ **IM**2命令が実行されているときにだけこの機能を有しますが、**MSX** は **IM**1命令を使用しているためここでは触れません。通常はプログラマが意識する必要はありません。

Rレジスタ

ダイナミック**RAM**というメモリの内容を保持するために用意されています。値はハードウェアと密接な関係があり、めまぐるしく変化しているため予想がつきません。

IX、IYレジスタ（16ビットインデックスレジスタ）

それぞれHLペアレジスタに近い機能を持っていますが8ビットレジスタに分解して使う事はできません。「**IX**+n」と書いて、「**IX**からnバイト目」という使い方ができますが、命令長が長くなるのが難点です。**IX**と記しましたが、Iレジスタ（割り込みベクタ）とは無関係です。

また区分上は専用レジスタですが、**CPU**自身の処理では、これらを書き換えたり、参照したりしませんから、汎用レジスタと同じものだと考えて構いません。

SPレジスタ（スタックポインタ）

スタックポインタと呼ばれ、スタックエリアのアドレスを持っています。

PUSH/POP、**CALL/RET**命令で非常に重要な意味を持ちますので後で詳

しく説明します。16ビット固定長レジスタです。

ＰＣレジスタ（プログラムカウンタ）

現在実行している命令のアドレスを持っています。１つの命令の実行が完了すると、ＰＣは自動的に次の命令のアドレスにセットされます。これも独立したレジスタで、Ｃレジスタとは無関係です。16ビット固定長です。

3−2 フラグ

Ｆレジスタはフラグ（*Flags*）レジスタと呼ばれます。

Ｆレジスタのそれぞれのビットのことをフラグといい、命令実行後の状態を知るときに使います。フラグを利用することにより条件分岐、つまり状況に応じた処理の選択が行えます。Ｆレジスタは８ビットですが、その内の２ビットは使われていません。ですから、Z−80には６種類のフラグがあることになります。

ここではフラグレジスタの構成およびそれぞれのビットの意味を説明します。

Ｆレジスタの構造（それぞれ１ビットを示す）

S	Z	未定義	H	未定義	P／V	N	C

Ｆレジスタ自身は８ビットの構成ですが、それぞれのビットに意味があります。

Ｓ（サインフラグ）

演算結果の最上位ビットに一致します。つまり、数値を２の補数表現とみなしたとき結果が負ならばＳ＝１、正またはゼロならばＳ＝０となります。正／負の判断に使われる事から、通常このフラグがセットされていることを「サインマイナス（M）」、リセットされていることを「サインプラス（P）」と言います。

Z（ゼロフラグ）

演算の結果Aレジスタの値が０の時にZ＝１、そうでなければZ＝０となります。

H（ハーフキャリーフラグ）

演算の中で、第３ビットから第４ビットへの繰り上げ、もしくは桁借りのあったときにH＝１となります。

P／V（パリティ／オーバーフローフラグ）

このフラグは２つの役割を兼ねていて、命令によってその意味が違います。パリティとはデータを２進数表現にしたときの１の個数の事で、オーバーフローとは「桁あふれ」（データの表現できる範囲を超えたこと）を意味します。

論理演算の時…パリティフラグとして働き、演算結果で各ビットのうち、１になっているものが奇数個であればP＝０、偶数個であればP＝１となります。ローテートシフト命令でもパリティフラグとして動作します。

算術演算の時…オバーフローフラグとして働き、これから演算をしようとするデータを２の補数による符号付き数とみなして計算し、その結果がオーバーフローすればV＝１、そうでなければV＝０となります。

N（減算フラグ）

直前の演算が減算であったときにN＝１となります。

C（キャリーフラグ）

加算の結果がオーバーフローしたとき、もしくは減算の際の引く数が引かれる数よりも大きかった場合にC＝１、そうでない場合はC＝０となります。ローテートシフト命令にも影響を受けます。論理演算があったときには必ずリセットされます。

このうち、Hフラグと Nフラグは極めて特殊で、１０進補正命令（**DAA**）以外ではまず使いません。反対に重要なのはS、Z、Cフラグで、これらは非常に頻繁に使用しますので、概略だけでも覚えておいて下さい。
Fレジスタ自身は８ビットの構成ですが、それぞれのビットに意味があります。

Simple ASM のマニュアル中の「**Coral Z−80 HANDBOOK**」には命令ごとにフラグレジスタの変化が書かれていますのでそちらも参考にして下さい。

3−3 プログラムの流れ

命令の長さ

Z−80の命令の形態は４種類あり、１バイトで表される命令を１バイト命令、２バイトで表される命令を２バイト命令、以下３バイト命令、４バイト命令と呼びます。

１バイト命令

メモリの中の１つの番地のメモリ（８ビット）を使う命令です。

２バイト命令

連続した２つの番地にまたがった命令で２バイト目には主にデータが入ります。

３バイト命令

連続した３つの番地にまたがった命令で、２バイト目、３バイト目は主に番地情報や２バイトのデータが入ります。

４バイト命令

４バイトの命令の最初の２バイトが命令、残りの２バイトには主に番地またはデータが入ります。

プログラムの流れ

もしプログラムに分岐命令がないときには、小さい番地から実行します。次に実行される番地はPCに記憶されます。今、PCの値をE000Hで、E000H番地の内容が２バイト命令の時は、PCの値がE002Hになります。続けて、E002H番地の内容が３バイト命令の時は、PCの値がE005Hとなります。　このようにPCの値は命令の長さにより値が変わっていきます。分岐命令があったときは、全く別の値になります。

上の図のように、Z−80はPCの値に従って命令を実行していきますが、先ほど説明したように２バイト命令、３バイト命令の２バイト目以降にはデータや番地情報が入っています。Z−80は命令、データの区別なしにPCの値をもとに忠実に実行しますので、万が一PCの値が指す番地に命令ではなくデータが入っていたときには。そのデータを命令として解釈してしまうため予想もつかない状態になってしまいます。上の図の例ですと、E000、E002、E005、E006、

E007番地は必ず命令でないといけないことになります。

命令の読み出し／解読／実行

Z−80は、逐次処理形（命令を１つづつ実行していく）の**CPU**です。以下の手順の繰り返しによりプログラムを実行していきます。複数の命令を一度に実行するようなことはありません。

①**PC**（プログラムカウンタ）の示すアドレスから、データを持ってくる。

②そのデータが、どんな手続きをするものなのかを解読する。もしその命令を実行するために他にもデータが必要ならばそれも読み出す。

③解読した命令を実行する。

④**PC**の値を次の命令があるアドレスに変更する。

⑤①に戻る。

パソコンをリセットした直後には、**PC**が0000H番地にセットされます。「ＰＣのアドレスをここに変更しろ」という命令がない限り、**CPU**はアドレスの若いほうから順序よく処理を進めていきます。

暴走について

機械語のプログラムが難しいと言われるのは、“暴走”と呼ばれる状態におちいることがあるからです。

これは、命令を間違えて書き込んでしまったときや、先ほども述べたように、命令でない部分を実行してしまった時などに起こります。

このような“暴走”起こしてしまったときには、メモリの内容が書き換えられてしまったり、画面が乱れてしまったりします。この暴走を防ぐには確実なプログラムを作成するしかありません。また、機械語プログラムを実行する前には必ずソースプログラムを保存するように心がけましょう。

第4章

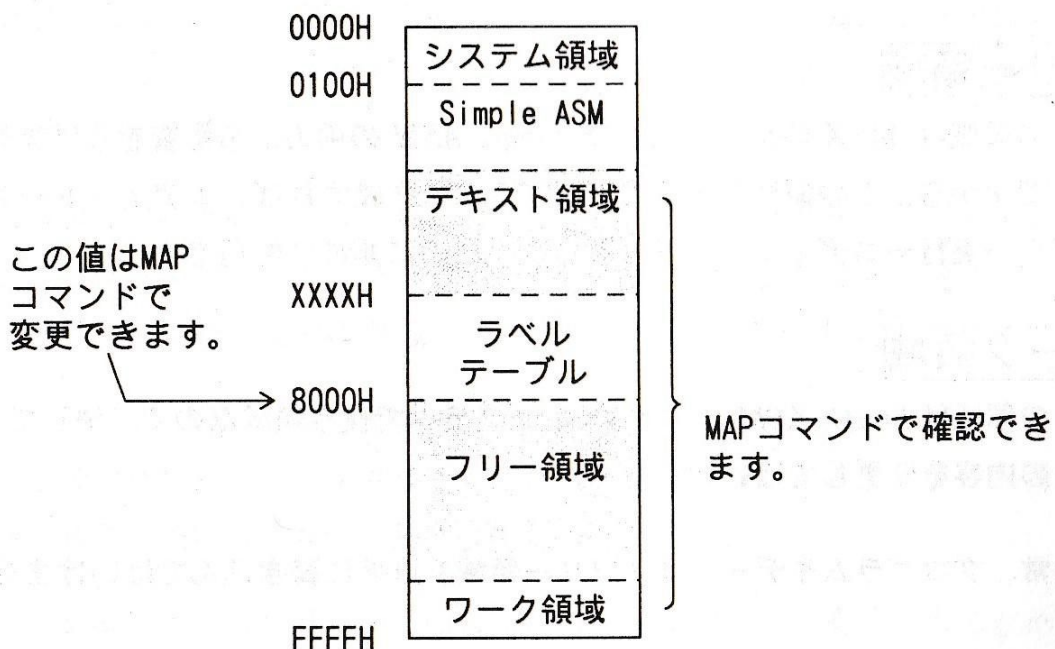
*Simple ASM*の基礎

4−1 Simple ASMを使う

本書は、Z−80の機械語命令を、サンプルプログラムを交えて実際に実行しながら学習していくものです。よって当然、アセンブラ、モニタなどのプログラム開発ツールが必要になります。もちろん、この本では株式会社コーラルのMSX用Z−80エディタアセンブラ *Simple ASM Ver.2.0* を使っています。

この *Simple ASM Ver.2.0* にはMSX−BASIC対応版とMSX−DOS対応版がセットされていますが、本書ではMSX−DOS版を例題に使っていますので、プログラミングの際にはMSX−DOS版 *Simple ASM* を起動して試してください。（1994年3月より *Simple ASM* は *Ver.3.0* に変わっていますが本書の内容は *Ver.3.0* のユーザーの方もそのままお読みください。）

4−2 メモリマップ

まず、メモリのどの部分が、どのように使われているかを再確認してみましょう。

このメモリマップはMSX−DOS版 *Simple ASM* が起動しているときの状態を表しています。

システム領域

この部分はMSX−DOSの基本的な部分が記録されており、さわることはできません。

Simple ASM

Simple ASM 自身が存在しています。

テキスト領域

この領域に、エディタで入力したアセンブラプログラムつまりソースプログラムが入ります。

ラベルテーブル

アセンブルを行ったときに、ラベルを登録するのに使います。エディタアセンブラのラベルテーブルは、テキスト領域で使われる最後の番地に続いて使われますので、テキスト領域の使用状況により、ラベルテーブルの開始番地は変わり、その境は、エディタのMAPコマンドで確認できます。

フリー領域

この領域はMSXのシステム及び *Simple ASM* の両方から影響をうけません。ですから、この領域に自分のプログラムを作成すれば、エディット→アセンブル→実行→エディットの繰り返しをするのに非常に便利です。

ワーク領域

この領域は *Simple ASM* 及びMSXのシステムで使う領域なので、決してメモリの内容を変更してはいけません。

通常、プログラムやデータは「フリー領域」以外に書き込んではいけません。

事前に**MAP**コマンドでフリー領域の範囲を確認しておいて下さい。その他の領域は**MSX**システム、**MSX−DOS**、*Simple ASM*が詳細に管理していますので、もし、フリー領域以外に書き込みを行った場合、あらゆる動作は保証されなくなります。最悪の場合はディスクが動き出して記録を破壊するようなこともあります。

4−3 Simple ASM環境内での注意事項

*Simple ASM*環境内というのは、**SA. COM**のプログラムを実行中ということです。機械語プログラムは別に**SA. COM**が起動されていなくとも実行することができるのですが、*Simple ASM*で機械語プログラムを開発するのには、*Simple ASM*で編集し、*Simple ASM*を終了させることなく、作成した機械語プログラムをテストしていきますので特に、*Simple ASM*環境内と限定しました。

ここでの注意事項は、**SA. COM**を実行している状態で機械語プログラムを作成するときの注意点になります。

①まず、プログラムを実行する前にはソースをセーブ（記録）する習慣をつけるということです。ソースプログラムのバグによりソースプログラムが消滅する可能性があります。

② *Simple ASM*上でＧコマンドによりオブジェクトプログラムを実行する場合、オブジェクトプログラムはフリー領域内にあることが必要です。通常はソースプログラムの先頭で**ORG**疑似命令により8000Hを指定します。

③プログラムを終了させるとき（アセンブラに制御を返すとき）は、**JP**命令により103H番地にジャンプさせてください。編集を続行することができます。**RET**命令を使用した場合、動作は保証されません。

④オブジェクトプログラム実行の直前にＡＯ コマンドを実行したことを確認

して下さい。Oのオプションをつけない A コマンドを実行したときには、オブジェクトプログラムはメモリ上に作成されていません。

4−4 デバッガ環境内での注意事項

機械語プログラムのテストをするには、SA. COM 内のモニタでテストするよりも、DB. COM を使ってテストした方が効率が良いときもあります。

ここでは、デバッガである DB. COM を使うときの注意事項について触れましょう。

①オブジェクトプログラムの作成は、フリー領域内に収まるようにしましょう。通常はソースの先頭で ORG 疑似命令により8000Hを指定します。

②プログラムを終了させるとき（デバッガに制御を返すとき）は、JP 命令によりBB03H番地にジャンプさせてください。RET 命令を使用した場合、動作は保証されません。

③ SA. COM の A コマンドでOオプションを指定して、オブジェクトプログラムをはメモリ上に作成しておきます。

④ *Simple ASM* 環境を抜けて、デバッガに入るときは次の手順に従います。

- 編集中のソースプログラムをセーブします。
  SA. COM を終了するとソースプログラムは消滅していまいます。
- Oオプションをつけた A コマンドを実行し、オブジェクトプログラムをメモリ上に作成します。
- SA. COM を終了します。SA. COM の SYS コマンドを実行します。
- デバッガ (DB. COM) を起動します。
  MSX−DOS コマンドラインから DB と入力して下さい。

4−5 COMファイルの作成方法

MSX−DOS コマンドラインから直接実行できるプログラムを開発するときは、まず **SA. COM** または、 **DB. COM** でバグのないプログラムを完成させます。

SA. COM または **DB. COM** で実行できるオブジェクトプログラム開始番地と **COM** ファイルの開始番地は異なります。また、プログラム全体を終了したときの処理も異なりますので変更します。

完成したと思われるソースプログラムの開始番地および、終了の処理を変更して再度アセンブルします。

①オブジェクトプログラムの開始番地の変更

オブジェクトプログラムの先頭を0100H番地に書き直す。（**ORG** 疑似命令）

②プログラムの最後をＲＥＴ命令に変更

SA. COM を使っているときのプログラムの終了処理は0103H番地へのジャンプ命令でした、そして、 **DB. COM** を使っているときはBB03H番地へのジャンプ命令でした。しかし、 **COM** ファイルの終了は **RET** 命令です。

例：

③ソースプログラムをセーブします。

④ **C** オプションをつけてアセンブル

C オプションをつけた **A** コマンドは、ファイル名の指定の際に拡張子を省略した場合、拡張子が **COM** となったファイルが作成されます。

もちろんこのとき **O** オプションをつけないでください。

第５章
*Simple ASM*操作法

本書では **MSX−DOS** 上で動作する *Simple ASM* を使ってアセンブラのプログラムの学習をしています。そのために **TAKERU** で購入したフロッピィディスクをもとにスムーズに学習できる環境を整える必要があります。

本章では購入したフロッピィディスクから開発／学習用のフロッピィディスクの作成方法を解説しています。

5−1 ハードウェア環境

まず *Simple ASM* を実行するためのハードウェア環境を確認します。

本体

Simple ASM は **CPU** は **Z−80**でも **R800**でも構いません。

メモリは64Kバイト以上必要です。

ＯＳ（オペレーティングシステム）

CPU と、ユーザとを結ぶ基本プログラムを指します。ここでは **MSX−DOS** を使いますので **MSX−DOS** を用意する必要があります。バージョンは1.0でも2.0でも構いません。

Simple ASM のパッケージにはセットされていませんので別途用意してください。

フロッピィディスクドライブ

MSX−DOS と *Simple ASM* はフロッピィディスクで供給されますので、必ず必要です。

Simple ASM は3.5インチ**2DD** でフォーマットされています。

ディスプレィ

ディスプレィは **MSX** 専用のディスプレイまたは家庭用テレビを使います。

MSX に接続できるテレビであれば、カラーでも白黒でも構いません。

プリンタ

プリンタは１行に80桁以上を印字でき、**MSX** に接続できるものを用意して下さい。

Simple ASM はプリンタがなくても使うことができますが、プログラム開発にはぜひ欲しいものです。

5−2 インストール作業

インストール作業とは、購入した *Simple ASM* と **MSX−DOS** のシステムを１枚のフロッピィディスク内におさめることを言います。

初めて使う場合のみこの作業が必要になります。

インストールにはつぎの３枚のフロッピィディスクが必要です。

- MSX-DOSあるいはMSX-DOS2のシステムディスク
- *Simple ASM* のディスク
- 実行用のフロッピィディスクとなる新しいフロッピィディスク

実行用のフロッピィディスクになる新しいフロッピィディスクには“*Simple ASM* 実行用ディスク”と記入したラベルを貼っておきましょう。以下の説明では実行用ディスクと呼びます。

今回のインストールの説明では **MSX−DOS2**のシステムディスクを使って説明していますが、**MSX−DOS** のシステムディスクを使ってインストール作業を行う人はCOMMAND2.COMとMSXDOS2.SYSがそれぞれCOMMAND.COM、MSXDOS.SYSになります。

インストールに必要なファイル

MSX-DOS2	MSX-DOS
MSXDOS2.SYS COMMAND2.COM	MSXDOS.SYS COMMAND.COM

ディスクドライブが１台のみの場合の手順

MSX−DOSの起動

・実行用ディスクとなる新しいフロッピーディスクを１枚用意します。通常は3.5インチ2DDを使って下さい。

・**MSX−DOS2**のシステムディスク（書き込みできないようにしておきましょう）をフロッピィディスク装置にセットして、リセットボタンを押して下さい。

・数秒後、いくつかのメッセージの後に **A>** と表示されたことを確認して下さい。（**DOS**のマニュアルも参照して下さい）

ＦＯＲＭＡＴの実行

・**MSX−DOS2**のディスクを抜いて、実行用ディスクを作るための新しいディスクをセットして下さい。

・**A>** に続いて、**FORMAT** と入力します。**DOS**のバージョンによってはいろいろと尋ねてくるので、自分にあった答えを入力します。

＜実行例＞

※＿＿＿の箇所はキーボードから入力

```
A>FORMAT ⏎
Drive name? (A,B) A

1 - 1 side, double track
2 - 2 sides,double track

? 2

All data on drive A: will be destroyed
Press any key to continue ⏎
```

この他の項目を要求してきた場合は**DOS**のマニュアルを参照して下さい。

・だいたい１〜２分ぐらいの時間が経った後、再び **A>** になったことを確認して下さい。

MSX－DOS2のシステムを転送

・FORMATしたディスクを抜いて、もう一度MSX－DOS2のディスクに交換して下さい。

ドライブ装置は１台ですが、MSX－DOS2のシステムディスクがＡドライブに割り付け、実行用ディスクがＢドライブに割り付けられた形で進めていきます。

ドライブＡとドライブＢは実際には同じドライブです。

パソコンが「Ａドライブにディスクをセットして下さい」という意味のメッセージが表示されたとき、ディスクドライブにはMSX－DOS2のディスクをセットします。反対にＢドライブを指定してきたときはインストール作業を行うための実行用ディスクをセットします。

・MSX－DOS2のシステムディスクに記録されている“MSXDOS2.SYS”と“COMMAND2.COM”のファイルを実行用ディスクに転送します。

<実行例>

```
MSX-DOS2のディスクをセットして、
A>COPY A:MSXDOS2.SYS  B: ⏎

Insert disk for drive B:
and strike a key when ready ⏎ ←ここで実行用ディスクをセット。
A>COPY A:COMMAND2.COM  B: ⏎

Insert disk for drive A:
and strike a key when ready ⏎ ←ここでMSX-DOS2のディスクをセット。

Insert disk for drive B:
and strike a key when ready ⏎ ←ここで実行用のディスクをセット。
```

*Simple ASM*のファイルを転送

・つぎにTAKERUで購入した*Simple ASM*のフロッピィディスクの中から２つのファイル“SA.COM”と“DB.COM”を実行用ディスクに転送します。
このフロッピィディスクも念のため書き込み禁止の状態にしておいてください。

<実行例>

```
A>COPY A:*.COM  B: ⏎

Insert disk for drive A:  ←ここでSimple ASM のディスクをセット。
and strike a key when ready ⏎

Insert disk for drive B:  ←実行用ディスクに入れ替えます。
and strike a key when ready ⏎
SA.COM                    ←アセンブラがコピーされました。
DB.COM                    ←デバッガがコピーされました。
     2 files copied
A>
```

確認

これでインストール作業は終了しました。念のため必要なファイルが記録されているか確認しましょう。　実行用ディスクをディスク装置にセットして、DIRコマンドを実行します。

```
A>DIR ⏎

Insert disk for drive A:  ←実行用ディスクをセットします。
and strike a key when ready ⏎
 Volume in drive A: has no name
 Directory of A:¥

MSXDOS2  SYS    xxxx xx-xx-xx xx:xxX
COMMAND2 COM    xxxx xx-xx-xx xx:xxX
SA       COM    xxxx xx-xx-xx xx:xxX
DB       COM    xxxx xx-xx-xx xx:xxX
 xxK in x files xxxK free
A>
```

まとめ

実行用ディスクに必要なもの

```
MSXDOS2  SYS （または  MSXDOS    SYS）
COMMAND2 COM （または  COMMAND   COM）
SA       COM
DB       COM
```

以上で、新しいディスクが *Simple ASM* 専用のフロッピーになりました。これからはこのディスクのことを「実行用ディスク」と呼びます。それでは実行用ディスクをセットしてリセットスイッチを押して下さい。

ディスクドライブが２台以上ある場合

この場合も、今、説明した流れと同様に実行用ディスクを作成することができます。ディスクドライブが２台あるとフロッピィディスクの抜き差しが無いだけスムーズに実行用ディスクを作成することができます。

5—3 実行用ディスクの起動

ＭＳＸ－ＤＯＳの起動

作成した実行用ディスクをドライブにいれて、リセットスイッチを押して下さい。

まず、最初に **MSX-DOS** が起動します。

```
MSX-DOS version 2.30
Copyright (1990) ASCII Corporation

A>  ←このことをプロンプトといいます。
```

バージョンによってはメッセージが幾分違います、また、機種によっては文字が大きく、メッセージが２行以上にまたがっていることもありますが、ここまでくれば大丈夫でしょう。

画面モードの変更

Simple ASM を使うには画面を80桁にした方が何かと便利です。 80桁にするにはMODEコマンドを使います。

```
A>MODE 80 ⏎
```

画面がクリアされて１行に80文字まで書けるようになりました。最初から80桁だった機種はクリアされただけですので気にしないで下さい。画面は広い方がプログラム開発に向いているので、このまま *Simple ASM* を起動しましょう。

また、80桁にして見づらくなった方は40桁にしておきましょう。

```
A>MODE 40 ⏎
```

ちなみに次のようにすると毎回指定した文字数で起動できます。

```
A>COPY CON AUTOEXEC.BAT ⏎
MODE 80 ⏎
^Z ⏎（これは、CTRLキーとZキーを同時に押して入力します。）
```

入力が終わってプロンプトがでたのを確認したら、リセットしてみて下さい。

アセンブラ／デバッガの起動

いづれの場合もコマンドラインから起動します。

```
A>SA  ←Simple ASM version 2.0 を起動します。
```

```
A>DB  ←付属のデバッガを起動します。
```

5―4 Simple ASMによる実習

ではここで、実際にソースプログラムの入力から実行までの流れを学びますよう。読みながら実際にパソコンを操作しましょう。

命令の関しては次章で解説しますのでここでは触れません。

それでは実習を始めましょう。ここで *Simple ASM* の使い方をマスターして下さい。

ソースプログラムの入力

・*Simple ASM* を起動して下さい。

・次のプログラムを入力します。プロンプト（>）の後から構わず丸写しして結構です。たとえば110行目はこうです。

```
>110  HAJIMEM:  EQU   8000H ⏎
```

入力するプログラム

```
110 HAJIME: EQU   8000H
120 OWARI:  EQU   0103H
130 ;
140         ORG  HAJIME
150         LD   HL,DATA
160 LOOP:   LD   A,(HL)
170         OR   A
180         JR   Z,EXIT
190         LD   C,02H
200         LD   E,A
210         PUSH HL
220         CALL 0005H
230         POP  HL
240         INC  HL
250         JR   LOOP
260 EXIT:   JP   OWARI
270 ;
280 DATA:   DEFB 'MSX '
290         DEFB 'Home '
300         DEFB 'Computer'
310         DEFB 0
320         END
```

・プログラム中のスペースは自由に取ることができますが、命令、ラベル名、レジスタ名などの中には取れません。（スペースを明確にするために、記号␣を使っています。）

```
例     LD HL,DATA
            ↓
 ×    LD HL,DA␣TA
```

・入力が終わったらリストを取って確認します。

```
>LIST ⏎
```

入力ミスが見つかった場合はここで修正します。

アセンブラの実行

・アセンブラを使って文法の間違いを探します。

```
>AU ⏎
```

ソースプログラムをアセンブルするＡコマンドに、パラメータＵを付けたものです。**U** は **Unlist** を意味し、エラー行のみを表示してくれます。エラーがなければ次のようになります。

```
>AU
PASS-1 } アセンブル作業の途中経過を表示します。
PASS-2
>
```

エラーがあれば、PASS-2の後にエラー行とエラー記号が示されます。

・エラーがないことを確認したら、ソースプログラムをディスクに記録します。

```
>SAVE "EX01" ⏎
```

・オブジェクトプログラムを作成します。

AO コマンドを実行して、メモリ上にマシンコード（つまり、マシン語プログラム）を作成します。

```
>AO ⏎
```

パラメータ **O** は、メモリ上にオブジェクトを作成することを意味します。また、パラメータ **U** を指定していませんのでアセンブルリストも表示します。

プリンタを持っている方は**AOP**と入力すればプリンタにも出力できます。

```
        アドレス（EQU 疑似命令は、とりあえずその値をここに示す）
110  8000               HAJIME: EQU   8000H
120  0103               OWARI:  EQU   0103H
130                     ;
        プログラムの始まりはここから（ORG によって決定された）
140                             ORG  HAJIME
150  8000 211680                LD   HL,DATA
160  8003 7E            LOOP:   LD   A,(HL)
170  8004 B7                    OR   A
180  8005 280B                  JR   Z,EXIT
190  8007 0E02                  LD   C,02H
200  8009 5F                    LD   E,A
210  800A E5                    PUSH HL
220  800B CD0500                CALL 0005H
230  800E E1                    POP  HL
240  800F 23                    INC  HL
250  8010 18F1                  JR   LOOP
260  8012 C30301        EXIT:   JP   OWARI
270                     ;
280  8016 4D535820      DATA:   DEFB 'MSX '
290  801A 486F6D65              DEFB 'Home '
     801E 20
300  801F 436F6D70              DEFB 'Computer'
     8023 75746572
310  8027 00                    DEFB 0
        プログラムは、ここまで
320  8028                       END
   アセンブラが、END命令は疑似命令のためマシンコードは何もない。
```

オブジェクトプログラムの実行

　モニタを使ってオブジェクトプログラムを確認します。アセンブルリストにより、オブジェクトプログラムの範囲が分かります。

```
>D8000,8027 ⏎
8000 21 16 80 7E B7 28 0B 0E  !...ｷ(..
8008 02 5F E5 CD 05 00 E1 23  ._.ﾍ...#
8010 18 F1 C3 03 01 C9 4D 53  ..ﾃ..ﾉMS
8018 58 20 48 6F 6D 65 20 43  X Home C
8020 6F 6D 70 75 74 65 72 00  omputer.
```

ということで、マシンコードが生成されていることを確認できました。

いよいよプログラムを実行できる段階になりました。走らせてみましょう。（念のため、念のため…　ディスクは外しておいた方がいいかも知れません。でたらめなデータを書き込んでしまう暴走も考えられますから。）

```
>G8000 ⏎
MSX Home Computer
>
```

これと違う結果が出た場合は、エラーということになります。ソースプログラムをよく確認して下さい。**MSX−DOS** 上の ***Simple ASM*** であること、***Simple ASM*** からの実行であることも確認して下さい。また、不幸にして暴走してしまった場合も、ソースプログラムはディスクに記録してありますから大丈夫です。いずれにしろ期待した結果が出なかった場合は、一度リセットした方が無難なときもあります。思わぬバグによりシステムが破壊されている可能性があるからです。

ＣＯＭファイルの作成

Simple ASM 上で期待通りの結果が得られたプログラムは、このままでは他の環境（**BASIC** や **ROM** など）で実行することはできません。今回は **DOS** のコマンドラインから実行できるプログラムに対応させてみましょう。まず、ソースプログラムの次の２行を書き直します。

修正前

```
100 HAJIME: EQU  8000H
260 EXIT:   JP   OWARI
```

→

修正後

```
100  HAJIME: EQU  100H
260  EXIT:   RET
```

以上でソースプログラムの変更は終わりです。ソースプログラムを変更したのですから当然再度アセンブルしてオブジェクトプログラムを作成する必要があります。

AOコマンドでオブジェクトプログラムをメモリ上に作成したり、Ｇコマンドでオブジェクトプログラムを実行してはいけません。

ここでは次のように入力して下さい。

```
>AC ⏎
PASS-1
PASS-2

Object filename? EX01.COM ⏎
```

拡張子は必ず .COMでなければなりません。拡張子を省略した場合は .COM が自動的に付加されます。

アセンブルリストが表示され、ディスクに上に“EX01.COM”が作成されます。エラーがなければ、ソースリストをセーブしておきます。

DIR コマンドで“EX01.COM”が作成されたことを確認して下さい。

Simple ASM を終了させて、**MSX−DOS** の状態にします。

```
>SYS ⏎
```

それではあなたの作った **DOS** コマンドとして実行してみましょう。

```
A>EX01 ⏎
MSX Home Computer
A>
```

いかがですか。*Simple ASM* の環境内と全く同じ動作をしたここと思います。以上の手順を踏むことにより、あなた自身で **DOS** コマンドを増やしていくことができるようになります。本書で基礎を学び、将来たいへん有効なプログラムを作成されることを期待します。

第6章
Z-80 命令のすべて

いよいよ本章ではZ–80の命令について学習していきます。実習形式で進めていきますので実際に操作することによって、理解を早めることができると思います。

6–1 命令の分類

いよいよこれからZ–80の命令をひとつひとつ説明していきます。Z–80の命令は非常にたくさんありますが、グループ別に分けるとそれほどでもありません。また、すべての命令を覚えなければプログラムを作成できないと言うわけでもありませんので、余りの多さにめげないでコツコツと理解してください。

また、本書では限られたスペースしかありませんので、すべての命令を詳細に解説しているものではありません。解説している命令以外を使用したいときには*Simple ASM*のマニュアル中にある「**Z–80 HAND BOOK**」を参考にしてください。

まず、Z–80の命令を分類してみましょう。

①８ビット転送命令
②16ビット転送命令
③交換命令
④ブロック転送命令
⑤８ビット算術論理演算
⑥16ビット算術演算
⑦ローテート／シフト命令
⑧ CPU コントロール命令
⑨アキュムレータ操作命令
⑩ジャンプ命令
⑪コール／リターン命令
⑫ビット操作命令

⑬ブロックサーチ命令
⑭入力／出力命令
⑮出力命令

また、これらの命令以外に疑似命令があります。
では、「５－４　*Simple ASM* による実習」のところで紹介したプログラムをもとにどのような命令を使っているかをみてみましょう。

```
110 HAJIME: EQU   8000H     } 疑似命令です。
120 OWARI:  EQU   0103H
130 ;                       - コメント行です 。
140         ORG  HAJIME     - これも疑似命令です。
150         LD   HL,DATA    - 16ビット転送命令です。
160 LOOP:   LD   A,(HL)     - ８ビット転送命令です。
170         OR   A          - ８ビット算術論理演算命令です。
180         JR   Z,EXIT     - ジャンプ命令です。
190         LD   C,02H      } ８ビット転送命令です。
200         LD   E,A
210         PUSH HL         - 16ビット転送命令です。
220         CALL 0005H      - コール命令です。
230         POP  HL         - 16ビット転送命令です。
240         INC  HL         - ８ビット算術論理演算命令です。
250         JR   LOOP       } ジャンプ命令です。
260 EXIT:   JP   OWARI
270 ;                       - コメント行です。
280 DATA:   DB   'MSX '     }
290         DB   'Home '    } 疑似命令です。
300         DB   'Computer' }
310         DB   0          }
320         END             - 疑似命令です。
```

こんな短いプルグラムでもいくつかの種類の命令を使っていることがわかります。

疑似命令とは、アセンブラがオブジェクトプログラムを作成する上で重要な役割を果たして言います。これも追って説明します。どんな疑似命令があるかは次章で説明しています。

6−2 命令の書き方

まず、命令をアセンブリ言語で書くときにはラベル、コード（ニーモニック）、オペンランド、コメントで構成されます。

たとえば次の１行をみてみると、

```
160 LOOP:    LD    A,(HL)
```

まず、**LOOP** という名のラベル名があります。続いて、**LD** というのは、コードになります。**A** および **(HL)** はオペランドになります。オペランドは２つありますので、それぞれ、第１オペランド、第２オペランドと呼ぶときもあります。

では、それぞれの意味を説明しましょう。

ラベル名

アセンブリ言語には **BASIC** 言語の **GOTO** 命令と同じように、“どこどこへ行け”という命令があります。これらの命令は行く先を指定しなければ行けません。このようなときに便利なのがラベル名です。

ラベル名がついていないときには16ビットの番地を直接指定する必要がありますが、ラベル名つければ、そのラベル名で指定できます。

たとえば、次の命令ですが、この命令はジャンプ命令で、“ **OWARI** ”というところにジャンプしなさいという意味です。またこの命令自身にも“ **EXIT** ”というラベル名がついています。

```
260 EXIT:    JP    OWARI
```

コード（命令コード、OPコード）

コードとはズバリ“命令”のことです。命令コードといったり、**OP** コード

と呼んだりもします。この命令の中には疑似命令の“**EQU**”や“**ORG**”、“**END**”などもあります。

オペランド

これは、命令に付属するものです。命令によっては複数のオペランドが必要な場合があります。

たとえば、次の行は“**OR**”という命令に付属して“**A**”がオペランドになります。

```
  170          OR   A
```

オペランドが複数ある場合は“,（カンマ）”で区切ります。次の例はオペランドが２つある場合の例です。

```
  200          LD   E,A
```

コメント

コメントとはプログラムに書く注釈、すなわち、何を書いたのかを示す注意書きです。

コメントは行の先頭にあっても、行の途中にあっても構いません。“;（セミコロン）”の文字がでてきたら、その文字以降がコメントになります。

プログラムを見やすくするために、行の先頭に；だけをつけるときもあります。

6−3 8ビット転送命令

Z−80の基本となる命令です。転送命令はロード命令とも呼びます。この命令は８ビットのデータを移動する命令になります。

OP コードは **LD** です。この **LD** は **Load** という単語の2文字をとったものです。

LD 命令は、レジスタとレジスタまたは、レジスタとメモリ間で行われます。また、直接あるデータをレジスタにセットするときもこの **LD** 命令を使います。

レジスタ・レジスタ間

```
LD  r,r'      レジスタr'の内容をレジスタrに転送する。
```

まず、一番最初に学ぶこの命令は、レジスタr'の内容をレジスタrに転送する命令です。レジスタrまたはr'は実際には8ビットレジスタの **A**、**B**、**C**、**D**、**E**、**H**、**L** のいずれかになります。

たとえば、**A** レジスタの内容が8だったとき、**B** レジスタの内容も **A** レジスタと同じ内容、つまりこのときは8にするのには“LD　B,A”となります。

例

```
LD  B,A       Aレジスタの内容をBレジスタに転送する。
```

この命令を実行しても **A** の内容はそのまま残っています。つまりここで使う転送という言葉は複写という意味です。

レジスタ・メモリ間

```
LD  r,(HL)  HLで示す番地の内容をレジスタrに転送する。
LD  (HL),r  レジスタrの内容をHLで示す番地に転送する。
```

次に、メモリとレジスタ間でのデータのやりとりです。メモリは16ビットのアドレスで表されますが、このアドレスを指定するのに **H**、**L** レジスタを組み合わせて使います。

このレジスタとメモリ間の移動はレジスタからメモリへデータを移動させる命令と、メモリからレジスタにデータを移動する命令の2種類があります。

例

```
LD  A,(HL)  HLで示す番地の内容をAレジスタに転送する。
LD  (HL),B  Bレジスタの内容をHLで示す番地に転送する。
```

レジスタ・データとメモリ・データ

```
LD  r,n        レジスタrにデータnを転送する。
LD  (HL),n     データnをHLで示す番地に転送する。
```

この命令は、レジスタに直接8ビットのデータをセットしたり、メモリに直接8ビットのデータセットしたりするときに使います。

例

```
LD  A,12H       Aレジスタに12Hをセットする。
LD  (HL),15H    HLで示す番地に15Hを書き込む。
```

ここで実習です。実際にMSXを使って命令の動作を確認します。

＜実習1＞

Aレジスタに8をセットするプログラムを作成し、結果を確認しなさい。

この簡単な命令を *Simple ASM* を使って行ってみましょう。

①ソースプログラムの消去

すでにメモリ内に何らかのプログラムがある場合、最初にメモリ内のプログラムを消去する事から始めましょう。

```
>NEW
```

②ソースプログラムの検討

“**A**レジスタに8をセットする”という命令は“LD　A,8”になります。ただ、この一行を書いただけではプログラムとして成り立ちません。では、実際にはどのようにしたらよいのでしょうか？

結果からみてみましょう。

```
100        ORG    8000H     ←  プログラムの作成位置の指定。
110        LD     A,8       ←  実際に必要なプログラム。
120        JP     0103H     ←  プログラム終了時にどうする？
130        END              ←  プログラムはもう終わり。
```

になります。100行、120行、130行は *Simple ASM* を使う上での決まった約束事と思って忘れずに書いてください。

③ソースプログラムの入力

１行入力後は必ず⏎キーを押してください。１画面に入る長さの行数ですのでわざわざ **LIST** コマンドで確認する必要はないでしょうが、長いプログラムの場合、**LIST** コマンドで入力したプログラムを確認するとよいでしょう。

```
100         ORG  8000H
110         LD   A,8
120         JP   0103H
130         END
```

④アセンブル

```
>AU
```

パラメータ **U** をつけるとエラー行が表示されます。もしここで、行番号とエラー記号が出た場合は、入力したソースプログラムををよく確認して修正して下さい。

⑤機械語（オブジェクトプログラム）の作成

```
>AO
PASS-2
100                 ORG  8000H  ←  フリーエリアの先頭です。
110  8000 3E08      LD   A,8    ←  マシンコードと見比べて下さい。
120  8002 C30301    JP   0103H  ←  Simple ASM に戻ります。
130  8005           END         ←  プログラムの終わりを示します。
```

パラメータ **O** をつけるとオブジェクトプログラムが作成されます。先ほどのパラメータ **U** と組み合わせて、１回で済ませてしまうこともできます。ただし。エラーが表示されたときのオブジェクトプログラムは実行できません。

⑥レジスタの確認

いよいよプログラムの実行ですが、今回は **A** レジスタにデータをセットするだけですからプログラムが正常に終了したとしても画面は変わりません。従

って *Simple ASM* のレジスタを見る機能を使って確認します。

```
>X
A  F  B  C  D  E  H  L
00 00 00 00 00 00 00 00
A' F' B' C' D' E' H' L'
00 00 00 00 00 00 00 00
IX   IY   SP   PC
0000 0000 F9F5 0000
```

まず、プログラム実行の前の状態を確認しておきます。これには **X** コマンドを使います。

X コマンドは、現在の全レジスタの内容を表示します。*Simple ASM* 起動直後は **SP** レジスタを除いて全て00になっていますが、上の例と違っていても構いません。今回は **A** レジスタへのロード命令ですから **A** レジスタの値を覚えておいて下さい。

⑦オブジェクトプログラムの実行

G コマンドを使ってプログラムを実行します。プログラムの実行開始番地は **ORG** 命令で指定した8000H番地です。終了番地は103番地です。もし、何も指定しないと、実行結果を表示させることはできません。これは、 *Simple ASM* がプログラム終了番地（正式には停止番地ですが。）を指定したときに、その時のレジスタ内容を表示させるようになっているからです。

```
>G8000,0103  ←  停止番地(103H番地) を指定してプログラムを実行する
A  F  B  C  D  E  H  L
08 00 00 00 00 00 00 00
            :
```

ここで、プログラム実行前の **A** レジスタの値と、実行後の **A** レジスタの値を見比べて下さい。実行前の値がなんであろうと、ロード命令により **A** レジスタに決められた値が書き込まれたことが分かったと思います。

また、**A** レジスタ以外にデータをセットするときも同様です。例えば **B** レジスタに８をロードしたいのならば、LD　B,8とします。

＜実習２＞

次のプログラムを実行し結果を確認してください。

```
100  ORG   8000H
110  LD    A,12
120  LD    B,12H
130  JP    103H
140  END
```

この問題はオペランド覧に10進数を指定したときと、16進数を指定したときの違いを見るプログラムです。

先ほど＜実習１＞と同じ手順で行ってみてください。

＜実習３＞

A、B、C、D、E、H、Lのレジスタの内容をすべて16進数で55Hにするプログラムを作ってください。

先ほどは、レジスタに直接データをセットする方法を説明しましたが、ここでは、レジスタ・レジスタ間の転送命令を使った例です。

ここでも最初にプログラムを掲載しますので、そのプログラムをもとに解説しましょう。

```
100          ORG  8000H
110          LD   A,55H  ←  ここでAレジスタに55Hをセット。
120          LD   B,A    ←  Aレジスタの内容をBレジスタに。
130          LD   C,A    ←  同じようにCレジスタに
140          LD   D,A    ←           :
150          LD   E,A    ←           :
160          LD   H,A    ←           :
170          LD   L,A    ←           :
180          JP   0103H
190          END
```

実際に試してみれば、正しい結果になることがわかるでしょう。130行はLD C,Aの代わりにLD C,Bとしても結果は同じになります。これは120行のLD B,Aを実行した時点で、**A**レジスタも、**B**レジスタもともに55になったからです。

＜実習４＞

9000H番地に88Hをセットするプログラムを作ってください。

　レジスタ・レジスタ間でのデータ転送の次はレジスタとメモリ間でのデータ転送を行ってみます。

　レジスタ・メモリ間でのデータ転送を行うには、まず、何番地のデータを転送するのか（あるいは、何番地へデータを転送する）を決めるために、何番地という情報を持つ必要があります。番地情報つまりアドレスは16ビットですので16ビットの長さのレジスタが必要です。**Z−80**では、このようにアドレスを示すために**H**レジスタと**L**レジスタがよく使われます。**H**レジスタには指定するアドレスの上位番地、**L**レジスタには指定するアドレスの下位番地をセットします。

　つまり、今回の場合、9000Hというアドレスを**HL**レジスタで表し、**H**レジスタには90Hが、**L**レジスタには00Hをセットします。その後**LD**命令でデータの転送を行います。

```
100  ORG  8000H
110  LD   A,88H  ←  ここでAレジスタに88Hをセット。
120  LD   H,90H  ←  Hレジスタにアドレスの上位８ビットをセット。
130  LD   L,00H  ←  Lレジスタにアドレスの下位８ビットをセット。
140  LD   (HL),A ←  Aレジスタの内容をHLで示す番地にをセット。
150  JP   0103H
160  END
```

　このプログラムの“LD　(HL),A”の部分がメモリに対する転送です。**HL**にカッコがついているのは、**HL**が指すところと言う意味です。

では実際にプログラムを入力し実行して結果を確かめて見ましょう。

①ソースプログラムの入力

　これは、もう何回も行っているので特に問題はないでしょう。入力した後は**LIST**コマンドで確認してください。プログラムを入力する前に、前回入力し

たプログラムがあるときには **NEW** コマンドを実行して下さい。

②オブジェクトプログラムの作成

これも今まで通りです。 **AO** または **AOU** と入力します。

③プログラムの実行

今回のプログラムの場合、実行結果のレジスタの内容を確認するのではなく、メモリの内容を確認するので停止番地を特に指定する必要はありません。もし、指定したときは、**H**、**L** レジスタの内容を確認することができます。

```
>G8000,0103  ←  プログラムを実行し、レジスタ内容を確認する場合
A  F  B  C  D  E  H  L
88 00 00 00 00 00 90 00
A' F' B' C' D' E' H' L'
00 00 00 00 00 00 00 00
IX   IY   SP   PC
0000 0000 F9F5 0103
```

④メモリ内容の確認

今回のプログラムを実行した結果はメモリの内容を見なければわかりません。メモリを内容を見るには **D** コマンドを実行します。

```
>D9000,9000  ←  メモリ内容を確認する。
9000 88
```

D コマンドは確認するメモリの最初の番地と最後の番地を指定します。
最後の番地を省略すると80バイト分表示されます。

6−4 16ビット転送命令Ｉ

ここでは、16ビットのデータの転送命令を学習します。16ビットのデータ転送命令は“**LD**”命令と、スタックを利用した命令があります。ここでは、８ビット転送命令と同じような記述をする“**LD**”命令に絞って説明します。

16ビットというのは２バイトです。ここでの転送は**A**レジスタと**F**レジスタを組み合わせた**AF**レジスタ、同じように、**BC**レジスタ、**DE**レジスタ、**HL**レジスタが利用できます。また最初から16ビットレジスタである**SP**レジスタ、**IX**レジスタ、**IY**レジスタも利用できます。

本書では16ビットレジスタを**dd**で表します。この**dd**には、**AF**、**BC**、**DE**、**HL**、**SP**、**IX**、**IY**のいずれかになります。

レジスタ・レジスタ間

LD	dd, dd'	16ビットレジスタdd'の内容をddレジスタに転送する。

この命令は次の３個に限定されています。

```
LD  SP,HL
LD  SP,IX
LD  SP,IY
```

SP（スタックポインタ）レジスタの内容はよほどのことがない限りさわらない方が賢明です。下手にさわると暴走を招きます。ですから、この命令の実習は行いません。

AFレジスタと**BC**または**DE**レジスタのデータ転送の命令が無いことに疑問を持たれるかも知れませんが、後ほど説明するスタックポインタを利用した16ビット転送命令で解決しますので心配しないでください。

レジスタ・メモリ間

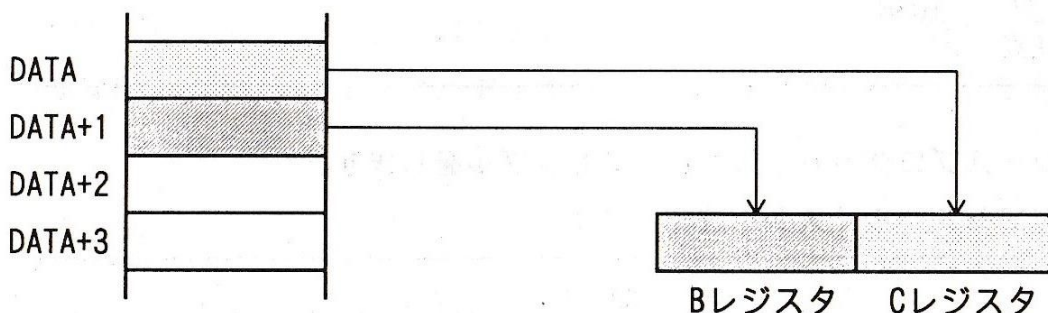
LD	dd, (nn)	nn番地、nn+1番地の内容を16ビットレジスタddに転送する。
LD	(nn), dd	16ビットレジスタddの内容をnn番地、nn＋１番地に転送する。

ここで説明する命令は　16ビットレジスタとメモリのデータ間での転送です。16ビットのデータを取り扱うわけですから、メモリ内のデータは２バイトのデータが対象になります。この２バイトのデータは指定した番地とその次の番地

になります。

例		
LD	BC,(9000H)	9000H番地、9001H番地の内容をBCレジスタにセット。
LD	(9000H),DE	16ビットレジスタDEの内容を9000H番地、9001H番地に転送する。

次の図は“LD　BC,DATA”の命令を実行したときのデータの移動を示してみました。

この命令を実行するとDATA番地の内容が**C**レジスタに、DATA+1番地の内容が**B**レジスタに転送されます。

このようにレジスタ・メモリ間のデータ転送では、指定した番地の内容が16ビットレジスタの下位８ビットにセットされ、上位８ビットには指定した番地の次の番地のデータが転送されます。

AF、**BC**、**DE**、**HL**レジスタでの上位８ビットは**A**、**B**、**D**、**H**のレジスタで、下位８ビットとは**F**、**C**、**E**、**L**レジスタになります。

レジスタ・16ビットデータ間

LD	dd,nn	16ビットデータnnを16ビットレジスタddに転送する。

先ほどのレジスタ・メモリ間のときの表現は(nn)でしたがレジスタ・16ビットデータ間ではnnになっています。()が付いている場合は()の中の値が番地になり、その番地のデータが有効になるのです。

たとえば、“LD　BC,8000H”という命令は**BC**レジスタに8000Hをセットす

る命令になりますが、“LD　BC,(8000H)”とすれば、8000H番地の内容をBCレジスタにセットすることになります。仮に8000H、8001H番地の内容が1234HだとしたならばBCレジスタは1234Hになります。

例

```
LD   BC,1234H        BCレジスタに1234Hをセットする。
```

＜実習５＞

次のプログラムを実行し結果を確認してください。

```
100  ORG   8000H
110  LD    BC,8000H
120  LD    DE,(8000H)
130  JP    103H
140  END
```

①まずソースプログラムを入力し、アセンブルをします。

ここでアセンブルリストを確認します。

```
100                         ORG  8000H
110   8000 010080           LD   BC,8000H
120   8003 ED5B0080         LD   DE,(8000H)
130   8007 C30301           JP   103H
140   800A                  END
```

オブジェクトプログラムを作成すると、8000H番地から8009H番地までの内容が変化します。

②エラーがないことを確認して、プログラムを実行します。

```
>G8000,0103  ←  プログラムを実行し、レジスタ内容を確認する場合
A  F  B  C  D  E  H  L
00 00 80 00 00 01 00 00
A' F' B' C' D' E' H' L'
00 00 00 00 00 00 00 00
IX   IY   SP   PC
0000 0000 F9F5 0103
```

結果ですが**B**レジスタが80H、**C**レジスタが00H、**D**レジスタが00H、**E**レジスタが01Hになっていることがわかります。つまり、**BC**レジスタが8000H、そして**DE**レジスタが0001Hになっていることになります。

そこでちょっと注意して欲しいのが、**DE**レジスタの内容で、**D**レジスタが“00”、**E**レジスタが“01”ということです。つまり、**D**レジスタには8001H番地の内容が、そして、**E**レジスタには8000H番地の内容が転送されることになるのです。

＜問題１＞

左右のプログラムの結果は等しいか等しくないか？

```
LD  B,12H
LD  C,34H
```

```
LD  BC,1234H
```

＜問題２＞

では、次の結果は？

```
LD  BC,(1234H)
```

```
LD  A,(1234H)
LD  B,A
LD  A,(1235H)
LD  C,A
```

＜回答＞

＜問題１＞は全く同じ結果になります。

＜問題２＞では“LD　BC,(1234H)”を実行すると**B**レジスタには1235H番地

の内容が、Ｃレジスタには1234H番地の内容がセットされます。
従って右と左とでは実行結果は異なります。

6-5 マシンコードについて

ソースプログラムをアセンブルするとオブジェクトプログラムが作成されます。では、そのオブジェクトプログラムとソースプログラムの関係はどうなっているのかを見てみましょう。

16ビット転送命令のときに使用したプログラムを例に説明しましょう。

```
100                     ORG  8000H
110   8000 010080       LD   BC,8000H
120   8003 ED5B0080     LD   DE,(8000H)
130   8007 C30301       JP   103H
140   800A              END
```

このオブジェクトプログラムは、8000H番地から8009H番地までの内容がオブジェクトプログラムになります。一つ一つの番地に対応させて考えると次のような図になります。

8000	01
8001	00
8002	80
8003	ED
	⋮

まず、“LD　BC,8000H”のマシンコードは“010080”になります。 最初の“01”が“LD　BC,????”の命令になります。次の“0080”が16ビットデータになります。これを分かりやすく図解すると次のようになります。

機械語
下位データ
上位データ

下位データ、上位データの順になる。

次に、“LD　DE,(8000H)”のマシンコードは“ED5B0080”になります。最初の“ED”と“5B”が“LD　DE,(????)”の命令になります。そして続く“0080”が16ビットデータになります。これを分かりやすく図解すると次のようになります。

機械語
機械語
下位アドレス
上位アドレス

下位アドレス、上位アドレスの順になる。

先ほどの“LD BC,????”は３バイトで構成される命令ですので３バイト命令で、“LD DE,(????)”は４バイトで構成されるので４バイト命令になります。

また、２バイトのアドレスデータや２バイトの単なるデータもマシンコードとして生成されますが、いずれの場合も下位１バイトが先にマシン語化され、上位１バイトがそれに続きます。

メモ

Z-80では、16ビットのデータを扱うときや、アドレスデータを扱うときは、何かにつけて、上位８ビットと下位８ビットが逆転する。

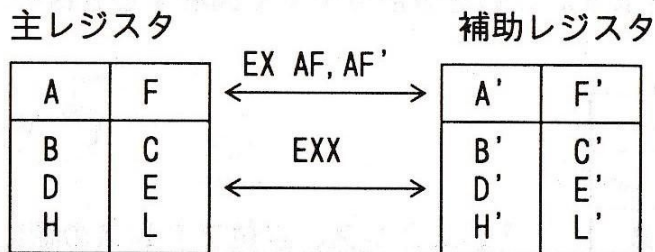
6−6 交換命令

EX AF,AF'、EXX命令

まず主レジスタと補助レジスタの内容を交換する命令です。

Ｚ−８０には主レジスタである　A、F、B、C、D、E、H、Lレジスタがありますが、この８個のレジスタと全く同じ内容のもがもう１セット用意されています。これを補助レジスタと呼んでいます。

この補助レジスタと主レジスタとを同時に使用することはできません。いろいろな演算をしたりするのには主レジスタだけになります。そこで、補助レジ

スタと主レジスタの内容を交換することにより、補助レジスタの内容使用することができるようになります。

この主レジスタと補助レジスタをを交換するための命令は２種類あります。まず一つは AF レジスタをまとめて交換するための“EX AF,AF'”と BCDEHL レジスタをまとめて交換するための“EXX”命令です。

B レジスタだけを交換するようなことはできません。

EX　DE,HL 命令

次は DE レジスタの内容と HL レジスタの内容をを交換するものです。

DE レジスタや HL レジスタは16ビットのデータを扱うときによく使うレジスタです。D レジスタや E レジスタを８ビットデータとして使うことよりも DE レジスタとしてペアで使うことの方が多いかも知れません。HL レジスタは確実に16ビットデータとして扱うことの方が多いものです。これは、HL レジスタを16ビットデータとして扱う命令が充実しているからに他なりません。

この命令は DE レジスタと HL レジスタとの内容を交換するものです。

6−7 ブロック転送命令

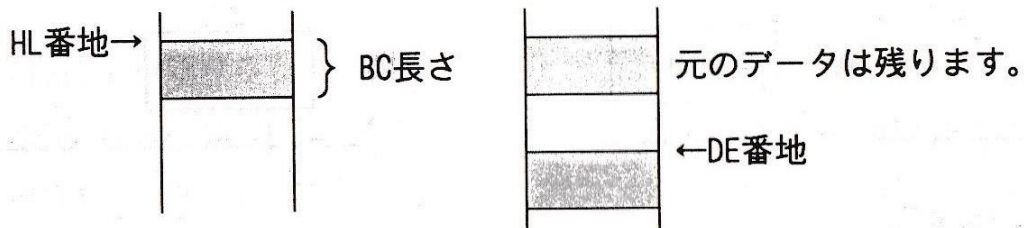
これは、メモリ上のデータをまとめて移動する命令です。この命令は次の４種類が用意されています。

```
LDI
LDIR
LDD
```

LDDR

これらの命令は **BC**、 **DE**、 **HL** レジスタが使われます。

BC：バイトカウンタと呼ばれるもので長さの指定に使います。
HL：転送元の開始番地に使います。
DE：転送先の番地指定に使います。

図にして見ると簡単にわかります。

LDI命令

LDI 命令は1バイトのみ転送されます。

命令実行後は **DE** レジスタと **HL** レジスタの内容が＋1され、 **BC** レジスタの内容は−1されます。

LDIR命令

この命令は、 **BC** レジスタの内容が0になるまで転送が行われます。つまり **BC** レジスタで示されるバイト数分が転送されます。

例

8000H番地から30Hバイト分を9000H番地以降に転送するときのレジスタの変化。

	実行前		実行後
DE	8000	DE	8030
HL	9000	HL	9030
BC	0030	BC	0000

<実習６>

LDIR 命令を使って実際にデータ転送命令を実行してみましょう。先ほどの例題そのものをプログラムして結果を見てみましょう。

プログラムは以下のようになります。

```
100  ORG   8000H
110  LD    BC,0030H
120  LD    HL,8000H
130  LD    DE,9000H
140  LDIR
150  JP    103H
150  END
```

①まずソースプログラムを入力し、アセンブルをします。

②エラーがないことを確認して、プログラムを実行します。

```
>G8000,0103  ←  プログラムを実行し、レジスタ内容を確認する場合
A  F  B  C  D  E  H  L
00 00 00 00 90 30 80 30
A' F' B' C' D' E' H' L'
00 00 00 00 00 00 00 00
IX   IY   SP   PC
0000 0000 F9F5 0103
```

③ **D** コマンドで8000H番地からの内容と9000H番地からの内容を比較してみましょう。

LDD 命令

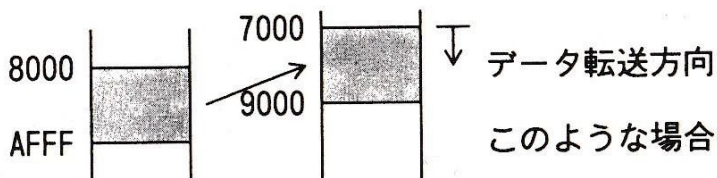
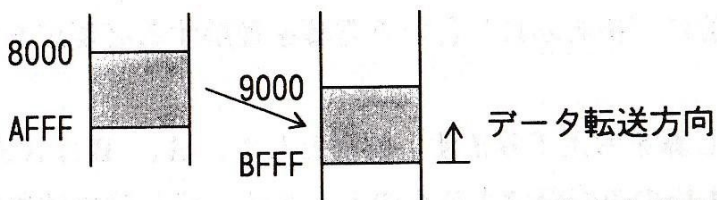
LDI 命令は１バイト転送ごとに **DE** レジスタと **HL** レジスタが＋１しましたが **LDD** 命令では−１されます。

LDDR 命令

LDDR 命令は **BC** レジスタが０になるまで、データ転送が繰り返されます。

LDI 命令と同じように１バイト転送ごとに **DE** レジスタと **HL** レジスタが−１されます。

LDIR 命令と LDDR 命令との使い分け

メモリ間のデータ転送には通常 **LDIR**、 **LDDR** 命令のどちらを使っても構いませんが、転送元の領域と転送先の領域が重なるときには注意が必要です。

このような場合は **LDDR** 命令を使う。

LDIR 命令を使った場合8000H番地のデータを9000H番地に転送したときに9000H番地のデータが失われてしまうためである。

メモ

インクリメントとデクリメントアセンブラを使っているとインクリメントという言葉とデクリメントという言葉がよくでてきます。しっかり覚えておいてください。
　　インクリメント ‥‥‥ ＋１することです。
　　デクリメント ‥‥‥‥ －１することです。

6－8 8ビット算術論理演算命令

“算術論理演算”という言葉がが突然でてきましたが、この言葉は２つに分解することができます。ひとつは“算術演算”で、もうひとつは“論理演算です。

算術演算‥‥加算と減算のことです。つまり足し算と引き算になります。

論理演算‥‥AND、OR、XORなどの論理演算です。

算術演算

キャリー

まず、算術命令を考える前に“桁あふれ”という言葉を理解する必要があります。

AとBとういデータの足し算を考えてみます。前提として、A、Bは10進数で表すものとして桁数は１桁のみ記録できるものとします。足し算の結果はCというところに記録します。このCも１桁しか表すことができません。

足し算は A＋B　→　C　という式で表します。

Aが“3”でBが“6”だとしたら、
3＋6　→　C　ですから　Cは“9”になります。

次にAが“3”でBが“7”だとしたら、

3＋7　→　C　ですから　Cは“10”になります。しかし、Cは一桁しか表すことができないので“0”になります。結果を“0”にするかわり、“桁あふれ”があったという情報が記録されます。

アセンブラの計算では、計算した結果に桁あふれがあるかどうかが重要になります。

10進数1桁で考えたことを16進2桁で考えても同じです。16進数2桁というのは8ビットになり、“00”～“FF”までを表すことができます。

8ビット同士の計算結果の範囲は“00”～“1FE”（“00”～“FF”、“100”～“1FE”）になります。計算結果も8ビットということにすると、下2桁の範囲では“00”～“FF”です。桁あふれは“1”のみです。

桁あふれ情報を“ある”場合“1”として“無い”場合を“0”にしても2種類です。つまり1ビットで表すことができます。

アセンブラの世界ではこれをキャリーといいます。このキャリーはフラグレジスタの中の1ビットを使っています。

例

16進数の足し算です。

```
   12      24      6A      F5      57      FF      87      34
 + 32    + 5A    + 9A    + 34    + 21    + FF    + 20    + 7A
 ----    ----    ----    ----    ----    ----    ----    ----
   44      7E     104     129      78     1FE      A7      AE
```

答えが3桁になっている 6A+9A と FF+FF のみがキャリーがたつ。

ADD命令

基本的な8ビットの算術演算命令です。

“**ADD　A,×**”と記述します。“×”の部分には**A**、**B**、**C**、**D**、**E**、**H**、**L**、**(HL)**、**(IX＋d)**、**(IY＋d)**およびデータを記述することができます。

A、**B**、**C**、**D**、**E**、**H**、**L**は各レジスタとの足し算になります。

(HL) は **A** レジスタとメモリのデータの足し算になります。**(IX＋d)**、**(IY＋d)** も同じく **A** レジスタとメモリのデータとの足し算になります。

計算された結果はすべて **A** レジスタに記録されます。桁あふれ情報はフラグレジスタに記録されます。

<実習７>

DATA1番地とDATA2番地とのデータを計算し、結果をDATA3番地に記憶させるプログラムを作りましょう。

```
100              ORG     8000H
110              LD      A,(DATA1)
120              LD      B,A
130              LD      A,(DATA2)
140              ADD     A,B
150              LD      (DATA3),A
160              JP      103H
170 DATA1:       DEFB    12H
180 DATA2:       DEFB    34H
190 DATA3:       DEFS    1
200              END
```

<プログラム解説>

100行目　プログラムの開始番地を8000H番地にしています。

110〜120行目

DATA1番地の内容をとりあえず **A** レジスタに持ってきて、それから **B** レジスタにコピーしています。これは、メモリのデータを直接 **B** レジスタに持ってくる命令がないからです。

130行目　**A** レジスタにDATA2番地のデータを転送しています。

140行目　**A** レジスタと **B** レジスタを足しています。結果は **A** レジスタに残ります。

150行目　Aレジスタに残った結果をDATA3番地に保存しています。

160行目　いつものように、プログラムが終了した場合は *Simple ASM* のコマンド待ちの状態にします。

170〜180行目

DEFB命令はその番地に１バイトのデータを用意する命令です。DATA1番地には12Hが、DATA2番地には34Hがセットされます。

190行目　DEFS命令は、記憶場所を確保する命令です。ここではオペランドが“1”になっていますので、１バイトのメモリが確保されたことになります。プログラムの実行前には何が入っているかわかりません。

プログラムの実行結果は、Dコマンドでメモリの内容を確認することによって確かめることができます。

＜実習８＞

先ほどのプログラムのDATA1とDATA2の値をいろいろ変えて試しましょう。

ADC命令

キャリーを含んだ足し算を行う命令です。

“**ADC　A, ×**”と記述します。**ADD**命令と同じように“×”の部分には**A**、**B**、**C**、**D**、**E**、**H**、**L**、**(HL)**、**(IX + d)**、**(IY + d)**記述することができます。さらに直接データを指定することもできます。

計算された結果は**A**レジスタに記録されます。**ADC**命令を実行した結果の桁あふれ情報はフラグレジスタに記録されます。つまり、実行前のフラグレジスタのキャリーフラグは前回計算したときの結果であり、実行後のキャリーフラグは今回の計算結果になります。

たとえば、“**ADC　A, B**”と“**ADD　A, B**”を比べた場合、次のように

なります。

ADC　A,B　…　A　＋　B　＋ Carry　→　Aレジスタ
ADD　A,B　…　A　＋　B　　　　　　→　Aレジスタ

＜実習９＞

＜実習７＞で行ったプログラムは桁あふれした結果を無視しましたが、今回は桁あふれを考慮したプログラムを考えてみます。

DATA1番地とDATA2番地の内容を加算して、その結果をDATA3番地及びその次の番地に保存しましょう。

```
100             ORG     8000H
110             LD      A,(DATA1)
120             LD      B,A
130             LD      A,(DATA2)
140             ADD     A,B
150             LD      (DATA3),A
160             LD      A,0
170             ADC     A,A
180             LD      (DATA3+1),A
190             JP      103H
200 DATA1:      DEFB    80H
210 DATA2:      DEFB    90H
220 DATA3:      DEFS    2
230             END
```

このプログラムは160行と170行がミソです。これは、次のような計算をしていることになります。

Aレジスタの内容　＋　Aレジスタの内容　＋ Carry　→　Aレジスタ

あらかじめ**A**レジスタの内容を“0”にしているのですから、この計算の結果はキャリーフラグが１のときのみ**A**レジスタが１になり、キャリーフラグが０のときは**A**レジスタも０になるのです。

プログラムの実行結果は**D**コマンドでメモリの内容を確認することによって確かめられます。

80Hと90Hの足した結果は110H（10進数ではないので170にはなりません。）ですので、DATA3番地には10Hが、DATA3番地の次の番地には01が記憶されていることが確かめられるはずです。

SUB命令

減算命令も加算命令と同じようにキャリーに影響されない減算とキャリーに影響される減算があります。

SUB 命令はキャリーを含まない計算です。減算した結果は **A** レジスタに残ります。

“**SUB**　×”と記述します。“**SUB**　**A**, ×”でないことに注意してください。

SBC命令

SBC はキャリーを含んだ減算です。減算した結果は **A** レジスタに残ります。

“**SCB**　**A**, ×”と記述します。

INC命令／DEC命令

INC 命令はインクリメント命令で **DEC** 命令はデクリメント命令です。
インクリメントとは＋１することで、デクリメントは−１することに他なりません。

INC 命令は“**INC**　×”、**DEC** 命令は“**DEC**　×”と記述します。
“×”は **A**、**B**、**C**、**D**、**E**、**H**、**L** の各レジスタと **(HL)** と **(IX + d)**、**(IY + d)** が指定できます。
DEC 命令も **INC** 命令と全く同じです。

CP命令

CP とはコンペアということで、**A** レジスタの内容テストするときに使います。“**CP**　×”と記述します。通常の **SUB** 命令と同じ演算をしますが、結果を **A** レジスタに保存しません。

たとえば　**SUB　B** の場合は **A － B** の計算をします。そして、フラグレジスタが変化し、**A** レジスタに **A － B** の結果が保存されます。しかし、**CP** 命令を使うとフラグレジスタのみが変化し **A** レジスタの結果は変わりません。

	内部の計算	フラグレジスタ	Aレジスタ
SUB　B CP　B	A－B A－B	変化する 変化する	A－B 元のまま

この命令は。たとえば **B** レジスタと **A** レジスタとを比較して、「結果が同じならば～しなさい。」などと言うときに便利です。

8ビット論理演算

論理演算には論理和、論理積、排他論理和の３つの種類があります。まず、論理演算について説明します。

最初に論理和、論理積、排他論理和の表を見てください。

論理和

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

論理積

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

排他論理和

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

この表の見方は **X** と **Y** が各値のときの演算結果がその右の欄に書かれています。たとえば論理和の表の中で **X** が０、**Y** が０のときの論理和の結果は０になるということです。

命令表のフラグの見方

フラグレジスタは実行した命令によって変化しますが、どんな命令を実行した後でも変化するかというとそんなことはありません。たとえばLD命令を実行してもフラグレジスタは変化しませんが、算術演算や論理演算の命令を実行したときは変化してしまいます。また、変化する内容はフラグレジスタの一部分のビットだったり、複数のビットだったりします。

どの命令を実行すればどのフラグが変化するかは Simple ASM のマニュアルの中にあるZ-80命令表を見ることにより簡単に確認することができます。ここでは命令表のフラグ欄の見方を解説しましょう。

８ビットロードのページの“LD　r,r'”を見てみましょう。

ニーモニック		フラグ						
		C	Z	P/V	S	N	H	
LD r,r'		●	●	●	●	●	●	

この“LD r,r'”命令のフラグ欄はすべて“●”になっています。この“●”は命令を実行してもフラグは変化しないことを表します。

続いて８ビット算術命令を見てみましょう。

ニーモニック		フラグ						
		C	Z	P/V	S	N	H	
LD r,r'		↕	↕	V	↕	0	↕	

C、Z、S、Hのところには“↕”マークが付いていますがこのマークは

結果によってフラグがたったり（１になること）、たたなかったり（０になること）します。

“V”はP／VフラグがVの意味としてフラグが変化することを表します。

最初から“0”になっているのは、この命令を実行すると無条件に“0”になることを表します。

このように命令表を見ればフラグが変化するのかどうかがすぐにわかります。

論理和

論理和は各ビットのいずれかが１のときに結果が１になり、両方とも０のときのみ結果が０になります。

論理積

論理積は各ビットともに１のときのみ結果は１になります。

排他論理和

いずれか一方のみが１のときのみ結果が１になります。

OR 命令／AND 命令／XOR 命令

論理和は OR 命令、論理積は AND 命令、排他論理和は XOR 命令を使います。

これらの命令は“OR　×”、“AND　×”、“XOR　×”と記述します。演算した結果は A レジスタに残ります。また、フラグレジスタの Z、P、S が変化します。C と N フラグは“０”になります。

＜実習10＞

DATA1番地の上位８ビットとDATA2番地の下位８ビットを合成したデータをANS番地にセットするプログラムを作成してみましょう。

```
100          ORG     8000H
110          LD      A,(DATA1)
120          AND     11110000B
130          LD      B,A
140          LD      A,(DATA2)
150          AND     00001111B
160          OR      B
170          LD      (ANS),A
180          JP      103H
190 DATA1:   DEFB    79H
200 DATA2:   DEFB    85H
210 ANS:     DEFS    1
220          END
```

＜プログラム解説＞

110〜130行

まず、DATA1番地のデータである79HをＡレジスタにセットし、1111000Bと**AND**を実行しています。実行結果は**B**レジスタに保存しておきます。

```
      0 1 1 1 1 0 0 1      ←――――― 79Hを２進数で表したもの
AND   1 1 1 1 0 0 0 0
----------------------
 =    0 1 1 1 0 0 0 0      ←――――― 結果は70Hとなった。
```

140〜150行

DATA2番地のデータ85Hを**A**レジスタにセットし00001111Bで**AND**命令を実行します。つまり、上位４ビットをマスクしたことになります。結果は05Hになります。

160行

DATA1番地の上位４ビットと、DATA2番地の下位４ビットを**OR**命令を使って合成します。

```
       0 1 1 1 0 0 0 0     ←――――― 70Hを２進数で表したもの
OR     0 0 0 0 0 1 0 1     ←――――― 05Hを２進数で表したもの
----------------------
    =  0 1 1 1 0 1 0 1     ←――――― 結果は75Hとなった。
```

170行

結果をANS番地に転送します。

このプログラムの実行結果は**D**コマンドでメモリの内容を直接確認することによって確かめられます。

DATA1番地の内容とDATA2番地の内容を適当に変えていろいろ試してください。

プログラムテクニック

マスク

AND命令を使ってマスクを行うことがよくあります。

１部のビットだけを“０”にすることをマスクするといいます。例えば８ビットデータの上位４ビットを０にし下位４ビットをそのまま生かすときは上位４ビットをマスクするといいます。Aレジスタの上位４ビットをマスクするには“AND　00001111B”命令を実行します。

逆に下位４ビットをマスクするには“AND　11110000B”命令を実行します。　“AND　00001111B”は“AND 0FH”で“AND　11110000B”は“AND 0F0H”と書いてもよいでしょう。

Aレジスタを０クリア

Aレジスタはよく使うレジスタですが、このレジスタをクリアするのに“XOR　A”命令を実行します。

これは排他論理和の「２つのビットのデータが同じときには０となる」という性質を利用したものです。Aレジスタ同士の排他論理和ですから当然すべてのビットが同じになるわけですから、演算結果は０になります。

Aレジスタをクリアするには他に“LD　A,0”や“SUB　A”命令もありますが、１バイト命令であることや実行した結果キャリーフラグが変化しないなどの理由から“XOR　A”命令を使います。

キャリーフラグのクリア

Aレジスタの内容をそのまま残し、キャリーフラグのみをクリアするには“OR　A”命令を実行します。

これはOR、AND、XORの各命令を実行するとキャリーフラグが０になるという性質を利用したものです。

6−9 16ビット算術命令

16ビットの演算には算術命令のみが存在し、論理命令は存在しません。16ビット演算は８ビットレジスタをペアで使う BC、DE、HL レジスタと最初から16ビットレジスタである SP、IX、IY レジスタを使います。

命令の種類	オペランドについて	計算の意味
ADD HL, ×	(×:BC、DE、HL、SP)	HL + × → HL
ADD IX, ×	(×:BC、DE、SP、IX)	IX + × → HL
ADD IY, ×	(×:BC、DE、SP、IY)	IY + × → HL
ADC HL, ×	(×:BC、DE、HL、SP)	HL + × + Carry → HL
SBC HL, ×	(×:BC、DE、HL、SP)	HL − × − Carry → HL
INC ×	(×:BC、DE、HL、SP、IX、IY)	× + 1 → ×
DEC ×	(×:BC、DE、HL、SP、IX、IY)	× − 1 → ×

足し算（加算）の命令にはキャリーフラグを含んだ命令と含まない命令がありますが、減算にはキャリーフラグを含まない命令はありません。従ってキャリーフラグを含まない計算を行うにはあらかじめキャリーフラグをリセットする必要があります。

＜実習11＞

DATA1番地、DATA1＋1番地に記録されている16ビットのデータとDATA2番地、DATA2＋1番地に記録されている16ビットのデータの加算と減算をするプログラムを作りましょう。

加算の計算結果はANS1番地に、減算の計算結果はANS2番地に保存することにします。

```
100        ORG     8000H
110        LD      HL,(DATA1)
120        LD      BC,(DATA2)
130        ADD     HL,BC
```

```
140              LD      (ANS1),HL
150              LD      HL,(DATA1)
160              OR      A
170              SBC     HL,BC
180              LD      (ANS2),HL
190              JP      103H
200     DATA1:   DW      1234H
210     DATA2:   DW      1111H
220     ANS1:    DS      2
230     ANS2:    DS      2
240              END
```

計算は **HL** レジスタと **BC** レジスタを使って行うことにします。

110行　**HL** レジスタにDATA1番地と次の番地の２バイトのデータをセットします。

120行　**BC** レジスタにDATA2番地と次の番地の２バイトのデータをセットします。

130行　**HL** レジスタと **BC** レジスタとの加算を行います。結果は **HL** レジスタに残ります。

140行　計算結果をANS1番地と次の番地に保存します。

150行　加算を行った際に **HL** レジスタはもとのDATA1番地のデータではなくなっているので再度DATA1番地と次の番地のデータをセットします。

160行　キャリーフラグをリセット('0')します。

170行　**HL** レジスタと **BC** レジスタとの減算を行います。結果は **HL** レジスタに残ります。

180行　結果をANS1番地と次の番地に保存します。

190行　プログラムを終了します。

200行　**DEFW** は２バイトのデータをメモリにセットする疑似命令です。ここではDATA1番地とその次の番地に1234Hのデータをセットしています。

210行　同じくDATA2番地と次の番地に1111Hのデータをセットしています。

220行　加算の結果を保存するために、２バイトのデータ領域を確保します。

230行　同じく減算の結果を保存するために、２バイトのデータ領域を確保します。

<プログラムの実行>

プログラムはいつものようにアセンブルし、“G8000”で実行します。レジスタの内容を見る必要がある場合のみ“G8000,103”とします。

<結果の確認>

DATA1、DATA2、ANS1、ANS2の番地はアセンブルリストからもわかるように、8017H、8019H、801BH、801DH番地ですから、**D**コマンドでメモリの内容を確認してみましょう。ANS2は２バイト確保していますので、メモリダンプの終了番地は801EH番地になりますので注意してください。

```
>D8017,801E
8017  34 12 11 11 45 23 23 01  4...E##.
```

LD命令でメモリとのやりとりを行うときには上位８ビットと下位８ビットが逆転することはすでに学んでいますので、そのことを念頭に入れてこの結果を確認します。

“DATA1番地とその次の続く番地”というのは長ったらしい言い方です。また16ビットデータというのは２バイトですので、ここでは単純に「DATA1番地の内容は～」と言うようにします。

つまり、DATA1番地の内容は1234Hで、DATA2番地のデータは1111Hになっています。計算結果はANS1番地が2345Hで、ANS2番地が0123Hになっています。

目的の結果が得られています。

では、DATA1、DATA2番地のデータを適当に変更していろいろ試してください。今回の例はあくまでも16進数で計算してることを頭に入れてください。

実行表を作る

プログラムの実行順に従ってレジスタがどのように変化しているか、メモリの内容がどのように変化していっているのかを表にしてみるとデバッグが楽にできます。一度試してください。

次の表は実行表のひとつの例です。

			HL	BC	ANS1	ASN2
110	LD	HL,(DATA1)	1234	????	????	????
120	LD	BC,(DATA2)	1234	1111	????	????
130	ADD	HL,BC	2345	1111	????	????
140	LD	(ANS1),HL	2345	1111	2345	????
150	LD	HL,(DATA1)	1234	1111	2345	????
160	OR	A	1234	1111	2345	????
170	SBC	HL,BC	0123	1111	2345	????
180	LD	(ANS2),HL	0123	1111	2345	0123

6−10 ローティト／シフト命令

ローティト命令、シフト命令という言葉を聞いてもどのような命令かピンとこないと思いますが、この命令は両方ともレジスタのデータをビット単位に操作するものです。ローティトとは何か？、シフトとは何か？を、最初にコマゴマと説明するよりも、一つ一つの命令を具体的に説明した方が分かりやすいと思いますのでさっそく説明に入ります。

ビットの呼び方

まず、２進数になおしたときのビットの呼び方を定義します。

ローティト命令にしてもシフト命令にしてもレジスタ内のデータを１ビットずらす命令です。

これらの命令はビットで考えなければ行けませんので、データはすべて２進数で表して考えます。

例えば16進数の99Hは２進数で10011001になります。
これを１ビットづつ分解すると次のようになります。

1	0	0	1	1	0	0	1

ただ、漠然と２進数を並べただけです。この２進数の各ビットに下位より０から順番を振ります。

この番号がビット位置になるのです。それぞれの番号にｂを付けて、どのビットかを表します。“b_7”は最上位ビットになります。

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	0	1	1	0	0	1

さらに、最上位ビットと最下位ビットだけを特殊な言い方として**MSB**、**LSB**と呼ばれています。

MSB							LSB
1	0	0	1	1	0	0	1

言葉の意味を理解したあらば、シフト命令から説明しましょう。

シフト命令

シフト命令はレジスタ内のデータを１ビットずらす命令です。

シフト命令は次の３種類があります。

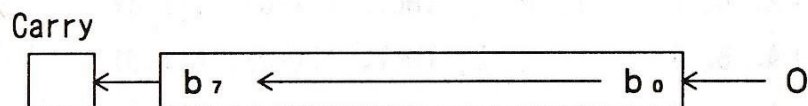
SLA　×（×：A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d)）
SRA　×（×：A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d)）
SRL　×（×：A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d)）

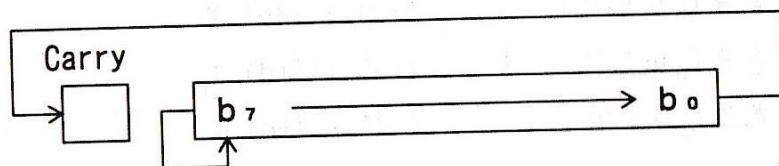
SLA　×

この命令はシフト・レフト・アリスメティックと呼びます。つまり真ん中の

レフトは左です。つまり、左に１ビットずらす命令です。
例えば、99Hを例に説明すると次のようになります。

では、もともと **MSB**（b7）のデータはどこに行ったのでしょう。また **LSB** はどうなるのでしょう。
この **MSB** と **LSB** がどうなるかによって後ほど説明するローテイト命令と異なるところです。
SLA 命令の時 **MSB** の内容はキャリーフラグにセットされます。そして、**LSB** には“0”がセットされます。　キャリーフラグと **LSB** のことを考慮すると次のようになります。

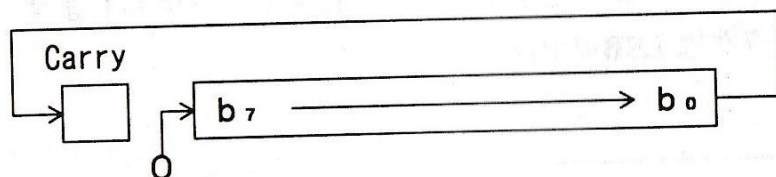
これを実行前、実行後の状態を一般的にあらわすと次のように図式できます。

SRA 命令

この命令は最初図式で表してみましょう。

この命令はシフト・ライト・アリスメティックと呼びます。つまり真ん中のライトは右です。つまり、右に１ビットずらす命令です。

図を見てわかるように、b_0の内容がキャリーフラグになります。そして、b_7のところはまたb_7になっています。

では、E5Hのデータで考えてみましょう。

SRL 命令

この命令も最初図式で表してみましょう。

この命令はシフト・ライト・ロジカルと呼びます。**SRA** と異なるは **MSB** の処理だけです。**MSB** は常に“0”がセットされます。

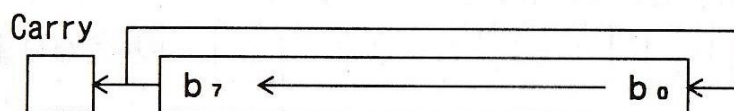
ローティト命令

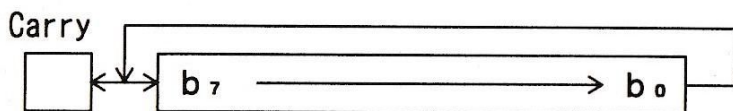
ローティト命令もシフト命令と同じようにレジスタ内のデータを１ビットずらす命令です。シフト命令と異なるのは **MSB** と **LSB** の処理です。

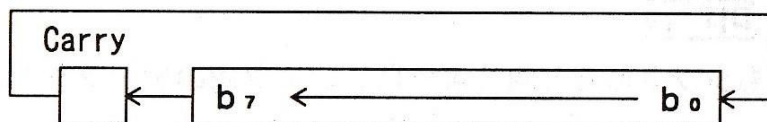
ローテイト命令は次の８種類があります。

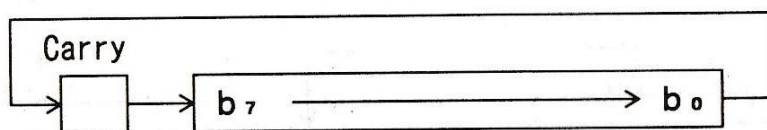
```
RLC  ×   ( × : A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d) )
RRC  ×   ( × : A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d) )
RL   ×   ( × : A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d) )
RR   ×   ( × : A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d) )
RLCA
RRCA
RLA
RRA
```

RLC 命令

ローテイトレフトサーキュラと呼びます。左に１ビットずらします。**LSB** とキャリーフラグは **MSB** の内容になります。

RRC 命令

ローテイトライトサーキュラと呼びます。右に１ビットずらします。**MSB** とキャリーフラグは **LSB** の内容になります。

RL 命令

ローテイトレフトと呼びます。キャリーを含んで左に１ビットずらします。

RR 命令

ローティトライトと呼びます。キャリーを含んで右に１ビットずらします。

RLCA／RRCA／RLA／RRA 命令

これらはすべて A レジスタを対象にした命令でそれぞれ、RLC　A、RRC　A、RL　A、RR　A と全く同じ働きをします。

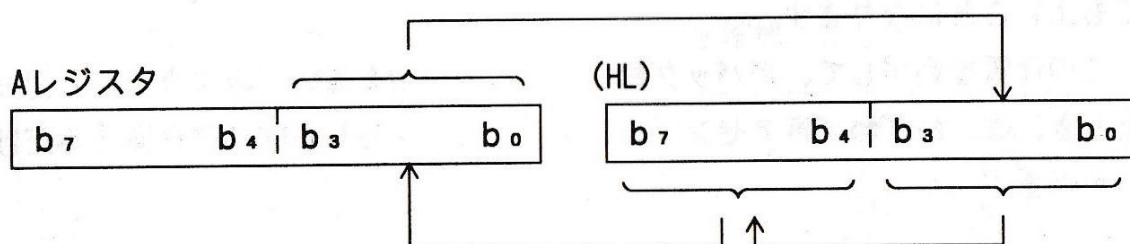
異なるのは１バイト命令であることと処理速度が速いということです。従って A レジスタのローティトするときにはこちらの命令を使用してください。

ローティトデジット命令

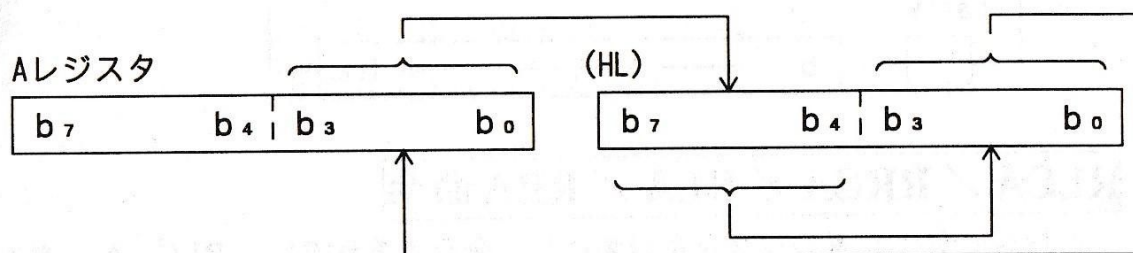
ローティト命令とシフト命令を説明しましたが、実はもう一つローティトデジット命令があります。

RLD 命令

ローティトレフトデジットと呼びます。これは、A レジスタの下位４ビットを HL レジスタが指すメモリの値とを４ビットごとにローティトすることです。

図にすると次のようになります。

RRD 命令

ローティトライトデジットと呼びます。これは、RLD とは逆に右側に４ビ

ットごとにローテイトします。
　図にすると次のようになります。

6−11 CPUコントロール命令

　Z−80AのCPUをコントロールする命令です。ハードウェアに関わってくる命令もありますからちょっと確かめようとするようなことはやめてください。
　ここでは簡単に触れるだけにとどめます。

NOP	CPUは何もしません。
HALT	CPUを停止します。
DI	割り込みを禁止します。
EI	割り込みを解除します。
IM 0、IM 1、IM 2	割り込みモードを変更します。

　この中でおもしろいのが“**NOP**”命令です。このマシンコードは00です。従ってメモリに00が連続していればすべて**NOP**命令になります。**NOP**命令は**CPU**は何も実行しません命令ですから、命令と命令の間にいくつ00があってもよいことになります。
　この性質を利用して、デバッグ最中に、この命令を省いてみようなどと思ったときには、わざわざ再アセンブルしなくとも、メモリを00に置き換えるだけですみます。

6−12 アキュムレータ操作命令

アキュムレータ操作命令は**DAA**、**CPL**、**NEG**、**CCF**、**SCF**命令があります。アキュムレータとは**A**レジスタとフラグレジスタのことです。

DAA命令

この命令は**A**レジスタの内容を10進数の２桁に補正するための命令です。

Aレジスタは00から**FFH**の値をとりますが、上位４ビット、下位４ビットを分離し、それぞれ０〜９までの値しかとらないようにします。つまり、１バイトで２桁の10進数を表すようにした考えです。

16進数２桁を10進数と見立てたとき

00H、01H、02H・・・09H、10H、11H・・・・89H、90H、91H、92H、93H・・・・99H

このように１バイトで10進数２桁を表すようにしたとき、加算命令を実行すると結果が10進数になりません。例えば72Hと18Hを加算したときに得たい結果は90Hなのに、通常ですと8AHになってしまいます。

そこで8AHというように結果を90Hという結果に変えるのが**DAA**命令です。このような処理を“10進数に補正する”と呼びます。

＜実習12＞

次の二つのプログラムを実行し、**DAA**命令を理解しましょう

```
100          ORG     8000H
110          LD      A,72H
120          ADD     A,18H
130          LD      (ANS),A
140          JP      103H
150  ANS:    DEFS    1
160          END
```

```
100          ORG     8000H
110          LD      A,72H
120          ADD     A,18H
130          DAA
140          LD      (ANS),A
150          JP      103H
160  ANS:    DEFS    1
170          END
```

CPL命令

この命令はＡレジスタのすべてのビットを反転させる命令です。

例えば、

```
LD    A,11001101B
CPL
```

を実行するとＡレジスタの内容は“00110010”になります。

NEG命令

この命令はＡレジスタの内容を２の補数にするものです。

例えば、

```
LD    A,5
NEG
```

を実行するとＡレジスタの内容は“11111011”になります。

補数というのはすべてのビットを反転して１を足すことはすでに学びました。では、５の補数を求めてみましょう。５は“00000101”でこれを反転すると“11111010”になり、これに１を足すと“111110111”になります。つまりこれが“−５”というわけです。プログラム結果と同じ回答が得られるでしょう。

CCF命令

キャリーフラグを反転します。

SCF命令

キャリーフラグを“１”にします。

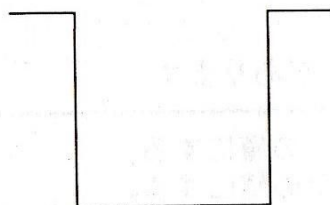
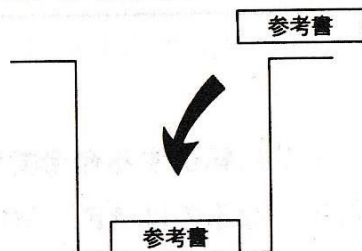
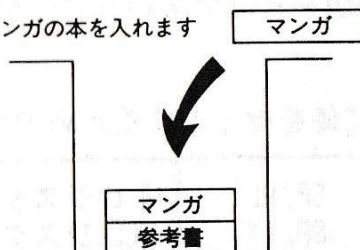
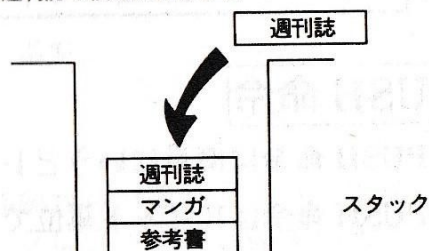
SCF 命令と CCF 命令を続けて実行することによりキャリーフラグが“０”になることがわかるでしょう。

6−13 16ビット転送命令II

Z−80の特徴であるスタック命令を使った16ビット転送命令を説明します。スタックポインタは理解しずらい雰囲気もありますが、理解すれば強力な武器になります。是非ともマスターしてください。

スタックの働き

スタックはデータの積み重ねをしていきます。また積み重ねしたものは、上からしか取り出すことができません。

次の図を見てください。

スタックの働きを分かりやすい図にしたものです。“参考書”、“マンガ”、“週刊誌”を箱の中に積み重ねた状態で考えましょう。

“参考書”、“マンガ”、“週刊誌”の順に積み重ねたもは“週刊誌”、“マンガ”、“参考書”の順でしか取り出すことができません。

メモリ上にデータを順番に積み重ねこと、積み重ねたデータを順番に取り出すこと、この働きがスタックです。

積み重ねることを **PUSH**（プッシュ）と呼び、積み重ねたデータを取り出すことを **POP**（ポップ）と呼びます。

また、積み重ねた位置を記憶するのがスタックポインタと呼ぶレジスタです。

SPレジスタについて

スタックポインタを記憶しているレジスタは **SP** レジスタです。スタックはメモリ上の一部を使いますので、あるメモリ番地を指していることになります。言い変えれば **SP** レジスタのデータは番地情報だということです。

私たちが *Simple ASM* を使ってプログラムを実行させせる場合は、スタックは初期状態から始めることになります。しかし、どうしても自分でスタックポインタの値を自分で決める必要があるときは **SP** レジスタにデータをセットします。

SP に値をセットするための命令は以下のものがあります。

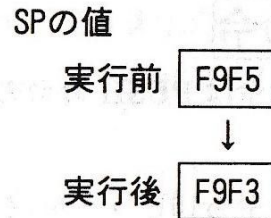
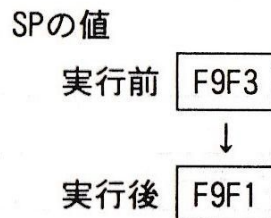
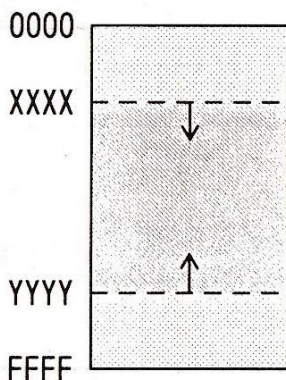
LD　SP,HL	HLレジスタの値をSPレジスタの値にする。
LD　SP,IX	IXレジスタの値をSPレジスタの値にする。
LD　SP,IY	IYレジスタの値をSPレジスタの値にする。
LD　SP,nn	データnnをSPレジスタの値とする。
LD　SP,(nn)	nn番地、nn+1番地のデータをSPレジスタの値にする。

PUSH命令

PUSH 命令は簡単にいうとレジスタの値をメモリに転送する命令です。

PUSH 命令は16ビット単位で行われます。対象となるのは **AF**、 **BC**、 **DE**、**HL**、 **IX**、 **IY** の各レジスタです。

この命令を実行するとレジスタの内容がスタックポインタが指すメモリ領域の一部に記録される作業と、**SP** の値を－２する作業が行われます。

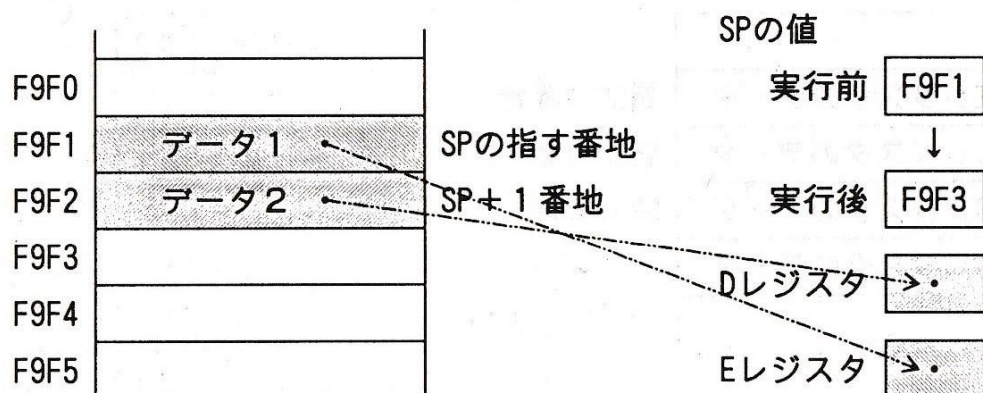
たとえば、 **SP** の値がF9F5番地（ *Simple ASM* の初期値）だったとき、**PUSH BC** を実行すると次のようになります。　記録される部分は **SP** の値の－１番地からです。

続けて、**PUSH　DE** を実行すると、次のように変化します。

PUSH 命令を実行するたびにスタックポインタが指す番地は２番地づつ少なくなってきます。つまり大きい番地から小さい番地の方にデータが記録されていくことになるのです。通常プログラムは、小さい番地から大きい番地の方に作成されますので図にすると次のようになります。

自由に使える領域がXXXX番地からYYYY番地だとしたならば、プログラムの開始番地をXXXX番地として、スタックポインタの初期値をYYYY+1番地とします。

POP命令

POP 命令は PUSH 命令の逆でメモリ上のデータをレジスタに転送する命令です。

POP 命令も16ビット単位で行われます。対象となるのは AF、 BC、 DE、HL、 IX、 IY の各レジスタです。

この命令を実行するとスタックポインタが指すメモリ領域のデータをレジスタに持ってきて、 SP の値を＋２します。

たとえば、 SP の値が F9F1番地だったとき、 POP　DE を実行すると次のようになります。

＜実習13＞

PUSH 命令と POP 命令を使って BC レジスタと DE レジスタの内容を交換するプログラムを作成しましょう。なお、スタックポインタの値は初期のままとします。

```
100          ORG     8000H
110          LD      BC,1234H
120          LD      DE,5678H
130          PUSH    BC
140          PUSH    DE
150          POP     BC
160          POP     DE
170          JP      103H
```

このプログラムのポイントは130〜140行と150〜160行です。 PUSH する順

番と **POP** する順番が同じときは、同じデータが戻ってきますが、**PUSH** する順番がと **POP** する順番を変えることによって **BC** レジスタと **DE** レジスタのデータを簡単に交換することができます。

<プログラムの実行>
アセンブルし、エラーがないことを確認してから実行します。
実行結果はレジスタ内容を見ることによって確かめることができますから“G8000,103”と指定します。

6−14 ジャンプ命令

プログラムは通常、番地の順に実行されますが、ジャンプ命令を使うことにより任意の番地にプログラムの流れを変えることができます。
今までにでてきたすべてのサンプルプログラムの最後に使っていた“JP 103H”命令もジャンプ命令です。
ここでは、そのジャンプ命令について学習しましょう。

無条件ジャンプと条件ジャンプ

ジャンプ命令は「どこどこ番地にジャンプしろ！」というものと「〜だったら、どこどこ番地にジャンプしろ！」というものに分けて考えられます。前者を無条件ジャンプ（ジャンプ命令を実行する条件が一切ない）と呼び、後者を条件ジャンプと呼びます。この条件ジャンプは、例えば「計算した結果が“0”ならばどこどこへジャンプしろ！」というように使います。

無条件ジャンプ命令

無条件ジャンプ命令は３バイト命令の“**JP** 命令”と２バイト命令“**JR** 命令”があります。

３バイト命令は“**JP**　アドレス”です。ここでのアドレスは16進で表されます。マシンコードは３バイト使われ、最初は無条件ジャンプの命令コードで、続く２バイトがジャンプ先のアドレスです。このアドレスは下位アドレス、上位アドレスの順になります。

C3
下位アドレス
上位アドレス

２バイト命令はリラティブジャンプと呼ばれます。日本語で言うと相対ジャンプです。

つまり、ジャンプ先のアドレスをジャンプ命令がある位置を元に「何バイト先へジャンプ」とか、「何バイト手前へジャンプ」というような使い方になります。

ジャンプ先は２バイト命令の１バイト目の前後－１２６～＋１２９番地までを指定することができます。

リラティブジャンプは２バイト命令なので使用するメモリが少なくて済みますので近い場所にジャンプするときには積極的にこの命令を使いましょう。
－126～ ＋129番地の範囲に入っているかどおか不明なときはとりあえずリラティブジャンプで記述し、アセンブルしてみます。もし、リラティブジャンプができない範囲ならばアセンブルのときにエラーが表示されますのでその時点でソースプログラムを変更すればよいでしょう。

２バイト命令のリラティブジャンプは“**JR**　相対位置”です。ここでの相対位置は１６進で表されます。

マシンコードは２バイト使われ、最初は無条件リラティブジャンプの命令コードで18Hになります。
続いて相対位置のデータです。これは２の補数表現になります。この相対位置のデータはアセンブラが自動的に計算してくれるのでいちいち計算する必要はありません。

条件ジャンプ

条件ジャンプ命令も３バイト命令とリラティブジャンプを行う２バイト命令があります。

どのような条件の種類があるのかを表にしてみました。

条件	条件内容	判断のためのフラグ
ＮＺ	ゼロでなければ（Zフラグが１でなければ）	Zフラグ
Ｚ	ゼロならば（Zフラグが１ならば	Zフラグ
ＮＣ	キャリーフラグがゼロならば	キャリーフラグ
Ｃ	キャリーフラグが１ならば	キャリーフラグ
ＰＯ	パリティが奇数の時	P／Vフラグ
ＰＥ	パリティが偶数の時	P／Vフラグ
Ｐ	０を含むプラスのならば	サイン（S）フラグ
Ｍ	マイナスならば	サイン（S）フラグ

これらの条件はすべてフラグレジスタの内容で判断されます。

命令は次のように記述します。

３バイト命令	２バイト命令 （リラティブジャンプ）
JP　NZ,nn	JR　NZ,e
JP　Z,nn	JR　Z,e
JP　NC,nn	JR　NC,e
JP　C,nn	JR　C,e
JP　PO,nn	JR　PO,e
JP　PE,nn	JR　PE,e
JP　P,nn	JR　P,e
JP　M,nn	JR　M,e

nn：ジャンプ先のアドレス

e　：相対位置

＜実習14＞

DATA番地から並べられたデータの合計を求め、ANS番地、ANS+1番地に記憶するプログラムを作成します。データの最後は０です。合計をするためのデータは１バイトですが結果は桁あふれする可能性がありますので２バイトの領域を用意します。

まず、どのレジスタをどの役割で使うかを検討します。

A レジスタ …… 多目的で使う。

BC レジスタ …… 合計するデータを **C** レジスタにセットし、16進数の加算命令“**ADD　HL,BC**”を行うために利用。
合計するデータは1バイトなので **B** レジスタを常に0。

DE レジスタ …… 計算するデータを持ってくる番地を保存。

HL レジスタ …… 合計の結果を保存。

```
100            ORG     8000H
110            LD      DE,DATA
120            LD      HL,0
130    LOOP:   LD      A,(DE)
140            CP      0
150            JR      Z,EXIT
160            LD      B,0
170            LD      C,A
180            ADD     HL,BC
190            INC     DE
200            JR      LOOP
210 ;
220    EXIT:   LD      (ANS),HL
230            JP      103H
240 ;
250    DATA:   DEFB    12H,34H,56H,3EH
260            DEFB    80H,80H,80H,80H
270            DEFB    0
280    ANS:    DEFS    2
290            END
```

110行　**DE** レジスタにDATA番地の値をセットします。

120行　合計の結果を保存するのに **HL** レジスタを使いますが、その内容をここで“0”にしておきます。

130行～200行までが繰り返し処理になっています。

130行　**DE** 番地の内容を **A** レジスタに転送しています。1回目のこの処理では**12H**がセットされます。

140行　**A** レジスタが“0”かどうかを比較しています。
もし“0”ならば **Z** フラグがセットされます。

150行　もしも、Zフラグがたっていたならば、EXIT番地にジャンプします。

160行　Bレジスタを“0”にしています。

170行　Aレジスタの内容をCレジスタに転送します。

160行、170行でBCレジスタに合計するデータをセットしています。

180行　合計の計算を行います。　HL + DE → HL

190行　合計するデータの番地をひとつ繰り上げます。

200行　LOOP番地にジャンプします。

220行　合計の結果をANS、ANS+1番地に保存します。

230行　プログラムを終了します。

250〜270行　合計するデータです。

280行　データが最後だということを表します。

290行　合計のデータを保存する領域を２バイト確保します。

プログラムを入力し、アセンブルし、エラーがないことを確認して、実行します。実行結果はDコマンドで確認できます。

ANS、ANS+1番地はアセンブルリストからわかるように、8021〜8022H番地ですので

```
>D8021,8022
```

で確認することができます。“02DA”になっていれば正解です。

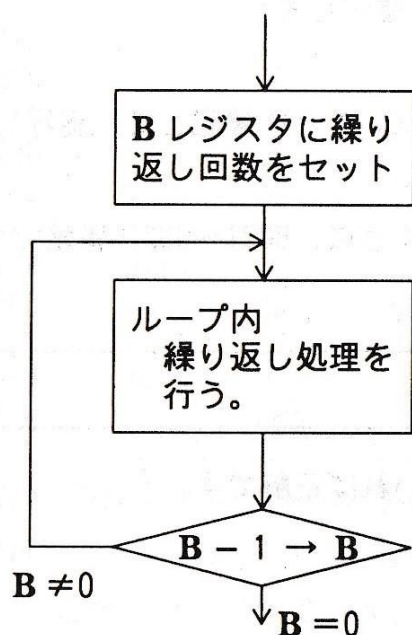
レジスタジャンプ命令

この命令はつぎの３種類があります。

JP　(HL)	HLレジスタの内容を番地としてジャンプします。
JP　(IX)	IXレジスタの内容を番地としてジャンプします。
JP　(IY)	IYレジスタの内容を番地としてジャンプします。

通常のジャンプ命令は直接、どこの番地にジャンプするかを指定しますが、この命令は、レジスタにセットされた値がジャンプ先になります。ですから、レジスタの値が思った値と異なった場合とんでもない番地にジャンプしてしまうことがありますので注意しなければ行けません。反面、この命令を使うことによって多方向分岐を行うことができます。

DJNZ（ループ回数制御）命令

Bレジスタのみ他のレジスタと違ってループ回数の制御に使うことができます。あらかじめセットされた**B**レジスタの内容を－１して、その結果が“0”ならば、次の命令を実行（ループからの脱出）します。“0”でなければ、オペランド覧に書かれた相対位置にジャンプします。アセンブラでは相対番地の計算が自動的にされますので、ラベルを書いておけばそのラベルの番地にジャンプします。

Bレジスタが0ならば次の処理

プログラムテクニック

DJNZ命令はBレジスタの内容を－1した結果を判断するので、1がセットしていれば繰り返し回数は1回ですが、最初に0がセットされていれば、繰り返し回数は256回になります。つまり、0を－1すると、FFHになり、それからFF回（255回）繰り返されるからです。

6−15 コール／リターン命令

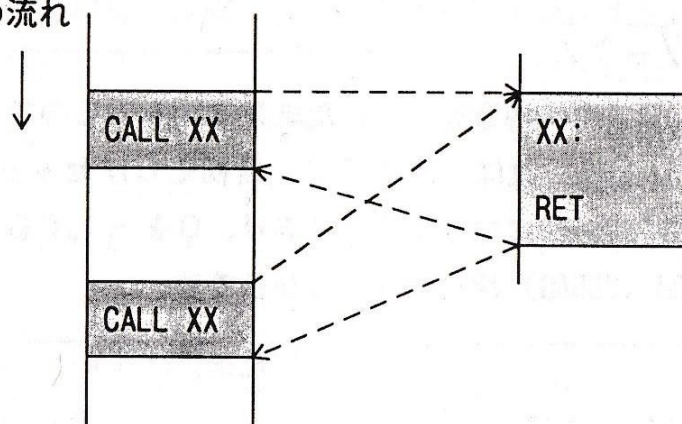
コール命令、リターン命令はいずれもサブルーチンを使う命令です。コール命令はサブルーチンを呼び出す処理で、リターン命令はサブルーチンから、サブルーチンを呼び出した処理に戻る命令です。

まずサブルーチンとは何かというのを説明しましょう。

サブルーチンとは

プログラムを実行していく上で同じような処理を繰り返し行うときには、その処理をひとまとめにします、そのひとまとめにした処理をサブルーチンと呼びます。

メインプログラムという言葉がありますが、これはプログラムの中心となるプログラムで、サブルーチンはこのメインプログラムから呼び出されます。また、サブルーチンから他のサブルーチンを呼び出すこともできます。

サブルーチンを呼び出す。つまり、サブルーチンプログラムを実行することを「サブルーチンをコールする」といいます。また、サブルーチンの処理を終え、呼び出されたプログラムに戻ることを「リターンする」といいます。

サブルーチン化する利点

機能ごとにサブルーチン化しておくと、プログラムのメンテナンス（修正）が簡単になります。また、メモリの節約にもなります。

CALL 命令

CALL 命令はジャンプ命令と同じように無条件 **CALL** 命令と条件 **CALL** 命令があります。

命令は次のように記述します。

CALL命令の種類
CALL　nn（無条件コール命令）
CALL　NZ, nn
CALL　Z, nn
CALL　NC, nn
CALL　C, nn
CALL　PO, nn
CALL　PE, nn
CALL　P, nn
CALL　M, nn

nn：ジャンプ先のアドレス

RET 命令

サブルーチン処理を終えるのは **RET** 命令です。この **RET** 命令も **CALL** 命

令同様、無条件 **RET** 命令と条件つき **RET** 命令があります。

命令は次のように記述します。

RET命令の種類	
RET	（無条件コール命令）
RET　NZ	
RET　Z	
RET　NC	
RET　C	
RET　PO	
RET　PE	
RET　P	
RET　M	

割り込み処理で使う
RETI、RETN命令も
存在します。

スタックについて

CALL 命令と **RET** 命令はスタックポインタの内容も変更されます。サブルーチンが呼び出される仕組み、サブルーチンからメインプログラムに戻る仕組みに興味がある方のみ参考にしてください。

CALL 命令の動作

PC（プログラムカウンタ）の内容がプログラムの実行番地になります。ジャンプ命令は、この **PC** の内容を変更することによって行われます。例えば9000H番地にジャンプするというのは、**PC** の値を9000Hに変更しているだけです。

CALL 命令も基本的に **PC** の内容を変更することによって行われます。同じように9000H番地の内容を **CALL** したときは **PC** の値を9000Hにすることです。

CALL 命令を実行したら、いずれ **RET** 命令を実行するわけですが、この **RET** 命令が実行されたら、**CALL** 命令を実行した次の番地に戻らなければ行けないということです。ジャンプ命令はどこに戻るかなどということを考えなくてよい分だけ単純です。

では、**CALL** 命令を実行したときの動作を見てみましょう。

CALL　nnを実行したとき

```
(SP) − 1  ← PCの上位番地 ┐ スタック領域に
(SP) − 2  ← PCの上位番地 ┘ 戻り番地をセット
SP        ← SP − 2
PC        ← nn             ― PCをサブルーチンの開始番地に
```

次に**RET**命令を実行したときの動作です。

```
PCの下位データ ← (SP)
PCの上位データ ← (SP) + 1
(SP)           ← (SP) + 2
```

CALL命令を実行したときの戻り番地はスタック領域に保存され、**RET**命令はスタック領域に保存された戻り番地に従ってサブルーチンを呼び出したプログラムに戻ります。

従って、スタック領域の内容が破壊されたり、**SP**の内容が変な値になったとき正しい場所に戻れなくなり暴走状態に陥ってしまいます。

＜実習15＞

DATA番地のデータを２進数として画面に表示するプログラムを作ってみましょう。

まず、メインの処理を考えます。

```
100              ORG     8000H
110      MAIN:   LD      A,(DATA)    Aレジスタにデータをセット
120              CALL    BIN         ２進数表示サブルーチン
130              JP      103H        プログラムの終了
140 ;
150      DATA:   DEFB    25H         表示したいデータ
160 ;
```

メインプログラムでは、DATA番地の情報を**A**レジスタにセットし、２進表示するサブルーチンをコールします。メインプログラムはMAINというラベルをつけます。**A**レジスタの内容を２進数で表示するサブルーチンにはBINというラベルをつけます。

サブルーチン“BIN”のプログラム

```
300     BIN:    LD      C,A
310             LD      B,8
320     BIN1:   RLA
330             LD      E,'0'
340             JR      NC,BIN2
350             LD      E,'1'
360     BIN2:   CALL    PUTCHR
370             DJNZ    BIN1
380     CRLF:   LD      E,0DH
390             CALL    PUTCHR
400             LD      E,0AH
410             CALL    PUTCHR
420             LD      A,C
430             RET
440 ;
```

このサブルーチンは、**A**レジスタの内容を２進数で表示するプログラムです。

サブルーチンプログラムを作成する上で、**A**レジスタの内容を変更する命令も使ってしまいます。そのため、サブルーチンの実行を終えてメインプログラムに戻ったときに**A**レジスタの内容が復元されるように、サブルーチンの最初の処理で**C**レジスタに一度転送しておき、サブルーチンの終了時に**C**レジスタから**A**レジスタに戻すようにしておきます。

Aレジスタは８ビットですので、“1”または“0”のいずれかの文字を８文字出力する必要があります。つまり、８回の繰り返しが必要です。

Bレジスタをこの８回のカウンタとして使うことにしました。**B**レジスタは**DJNZ**命令を使うことができるのでカウンタとして使うのに最適だからです。

“1”か“0”のいずれかを表示するのには**RLA**命令を使います。この命令を１回実行すれば、最上位ビットがキャリーフラグにセットされます。２回実行すれば、次のビットがセットされます。次々と繰り返せばすべてのビットをキャリーフラグにセットできるのです。

キャリーフラグがたっていたならば“1”を表示、キャリーフラグがたって

いなければ“０”を表示するようにします。プログラムでは、とりあえず“０”を表示データにセットし、その後キャリーフラグがたっているときには“１”を表示データにしています。

“CRLF”のラベルのところのプログラムは“0DH”と“0AH”コードを出力しています。これは、次の文字を表示させる前に改行する動作を行うためのものです。

サブルーチン“**PUTCHR**”のプログラム

```
500     PUTCHR: PUSH    AF
510             PUSH    BC
520             PUSH    DE
530             PUSH    HL
540             LD      C,02H
550             CALL    0005H
560             POP     HL
570             POP     DE
580             POP     BC
590             POP     AF
600             RET
610 ;
```

画面に１文字表示するには**MSX－DOS**のファンクションコールを利用します。**MSX－DOS**のファンクションコールについては後ほど詳しく説明しますが、１文字表示するファンクションコールは、**E**レジスタに表示する文字コードをセットし、**C**レジスタに“02H”をセットし、0005H番地をコールすることによって実現します。

ただしファンクションコールを実行すると、実行前のすべてのレジスタ内容は変わってしまいますので、必要なレジスタはスタック領域に保存しておきましょう。そして、ファンクションコールが終わった時点で復元するようにします。

今回は**A**レジスタ（２進表示させるデータ）と**B**レジスタ（ループカウンタ）と**C**レジスタ（最初の**A**レジスタの内容）さえ保存すればよいので

“PUSH DE”と“PUSH HL”は実行する必要はありませんが、他のプログラムからも利用できるようにすべてのレジスタを待避させておきました。

END 文はプログラムの最後に記述します。もしメインプログラムの最後に書いてしまうと、その後のサブルーチンプログラムはアセンブルされません。

```
800                END
```

リスタート命令

この命令は１バイトでコール命令を実行することができます。ハードウェアを設計する人はよく使いますが、私たちのように MSX のプログラムを作成する上では使いません。

リスタート命令	相当するコール命令
RST 0	CALL 0000H
RST 8	CALL 0008H
RST 16	CALL 0010H
RST 24	CALL 0018H
RST 32	CALL 0020H
RST 40	CALL 0028H
RST 48	CALL 0030H
RST 56	CALL 0038H

6−16 ビット操作命令

この命令は A〜L レジスタ、(HL)、(IX + d)、(IY + d)、の任意のビッ

トをセット(SET)、リセット(RES)、テスト(BIT)する命令です。

セットするというのはビットを"1"にすることで、リセットするというのはビットを"0"にすることです。

テストするというのは、あるビットが"1"か"0"かをテストする命令です。テストした結果は**Z**フラグに残ります。

```
SET  b,×   (×:A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d))
RES  b,×   (×:A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d))
BIT  b,×   (×:A、B、C、D、E、H、L、(HL)、(IX+d)、(IY+d))
```

例えば、**A**レジスタが最初"0000H"の時、"**SET 1, A**"、"**SET 3, A**"を実行する、**A**レジスタは次のように変わっていきます。

＜実習16＞

CALL命令のところで作成した2進数を表示するサブルーチンを利用して**A**レジスタが0のとき、"**SET 1, A**"命令の実行後、"**SET 3, A**"命令の実行後の様子を表示するプログラムを作成してみましょう。

メインプログラム（新規作成部分）

```
100             ORG     8000H
110     MAIN:   XOR     A
120             CALL    BIN
130             SET     1,A
140             CALL    BIN
150             SET     3,A
160             CALL    BIN
170             JP      103H
160 ;
```

SET 命令をひとつひとつ実行しているだけですので、プログラムの解説は特にいらないでしょう。

サブプログラム（前回作成したものをそのまま利用）

```
200     BIN:    LD      B,8
210     BIN1:   RLA
220             LD      E,'0'
230             JR      NC,BIN2
240             LD      E,'1'
250     BIN2:   CALL    PUTCHR
260             DJNZ    BIN1
270             RET
280 ;
310     PUTCHR: PUSH    AF
320             PUSH    BC
330             PUSH    DE
340             PUSH    HL
350             LD      C,02H
360             CALL    0005H
370             POP     HL
380             POP     DE
390             POP     BC
400             POP     AF
411             RET
420 ;
500             END
```

6−17 ブロックサーチ命令

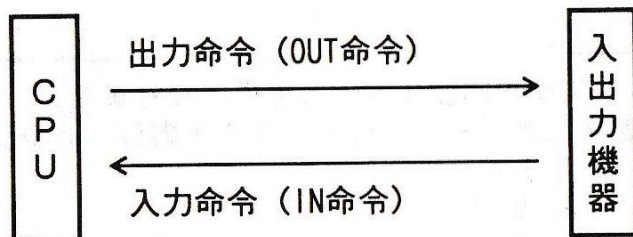
サーチというのは検索のことです。

この命令は、メモリの中から目的とするデータを探し出す命令です。探し出したいデータは A レジスタにセットしておき、どこの番地から探し始めるかを HL レジスタにセットしておきます。さらに、BC レジスタには何バイト探し出すかをセットしておきます。

ブロックサーチ命令は次の４種類があります。

CPIR	目的のデータを見つけだすまで、あるいはBCレジスタが0になるまでサーチを続けます。 探し出すデータはHLレジスタの番地からその後の番地の順になります。
CPI	CPIRは探し出すまで動作しますが、この命令は、HLレジスタの番地のデータが目的のデータかどうかを判断し、その後、HLレジスタを＋１、BCレジスタを−１します。もし目的のデータであれば、Zフラグがセットされます。
CPDR	CPIRはHLレジスタが順次＋１されていきますが、CPDR命令では−１されていきます。つまり番地の大きい順、言い換えれば、後ろから前の方向に検索していくことになるのです。もちろんBCレジスタはカウンタとして使うのですから、−１づつされます。
CPD	CPDはCPIと同じように１バイトのみの検索です。 HL、BCレジスタとも−１されます。

6−18 入力命令／出力命令

入出力命令は CPU と外部機器とのデータのやりとりを行うときに使います。

入出力機器というと大げさなものを想像するかも知れませんが、LSI が１個のときもあります。もちろんプリンタやフロッピィディスク装置の制御もこの入出力機器になります。

ただし、プリンタに文字を出力しようとしたとき、私たちが直接 OUT 命令を使ってプログラムを作ることはまずないでしょう。なぜならば、プリンタに文字を出力するようなサブルーチンがあらかじめ MSX に用意されているからです。このサブルーチンを使うことによって容易にプリンタに文字を印字できます。

IN 命令を使ったり、OUT 命令を使ったりすることは、MSX ではあまりありませんが、命令としては重要な命令ですので使い方だけ簡単に紹介しましょう。

ポート番号について

まず、入出力命令を理解する上で重要なのがポート番号です。メモリを考えると、ひとつひとつのメモリ（１バイト１バイトのメモリ）は個々にアドレス（番地）が付いています。このアドレスは１６ビットで表されることはもう理解できているはずです。

入出力はＩ／Ｏポートを介して行われますが、このＩ／Ｏポートにもひとつひとつ番地がふられています。ただし、メモリ番地と異なるのは、８ビットで表されるということです。なぜなら、Ｉ／Ｏポートは256個までしか使えな

いからです。

I／Oポートのアドレスは0～FFHになります。

入力命令

IN　A,n	nで示された入力機器のデータをAレジスタに入れます。
IN　r,(C)	Cレジスタの内容がポート番号になります。 指定されたポートからデータを入力し、rレジスタにセットします。 （r：A、B、C、D、E、H、L）
INI	Cレジスタの内容がポート番号になります。 ポートから入力されたデータをHLレジスタで示される番地にセットします。その後HLレジスタを＋1、Bレジスタを－1します。
INIR	INI命令の実行をBレジスタが0になるまで続けます。
IND	INI命令と同じですが、HLレジスタは－1されます。
INDR	IND命令の実行をBレジスタが0になるまで続けます。

出力命令

OUT　n,A	nで示された出力機器へAレジスタのデータを出力します。
OUT (C),A	Cレジスタの内容がポート番号になります。 指定されたポートへ、rレジスタのデータを出力します。（r：A、B、C、D、E、H、L）
OUTI	Cレジスタの内容がポート番号になります。 HLレジスタで示された番地の内容を出力します。 その後HLレジスタを＋1、Bレジスタを－1します。
OUTIR	OUTI命令の実行をBレジスタが0になるまで続けます。
OUTD	OUTI命令と同じですが、HLレジスタは－1されます。
OUTDR	OUTD命令の実行をBレジスタが0になるまで続けます。

6−19 R800命令

R800はMSX Turbo R 用に開発されたCPUです。Turbo Rを使用している人のみが使うことのできる命令をここでは説明します。

R800はZ−80の命令をすべて含み、さらに、いくつかの命令を追加したものですから今まで説明したすべての命令はそのまま使えます。

では、どんな命令が追加されたのかを一覧表で見てみましょう。表の中の16進数はマシン語でになります。

かけ算命令

	B	C	D	E	BC	SP
MUL ×	EDC1	EDC9	EDD1	EDD9	EDC3	EDF3

８ビット転送命令

	XH	XL	YH	YL
LD B, ×	DD44	DD45	FD44	FD45
LD C, ×	DD4C	DD4D	FD4C	FD4D
LD D, ×	DD54	DD55	FD54	FD55
LD E, ×	DD5C	DD5D	FD5C	FD5D
LD A, ×	DD7C	DD7D	FD7C	FD7D

	B	C	D	E	A	XH	XL	YH	YL	n
LD XH, ×	DD60	DD61	DD62	DD63	DD67	DD64	DD65			DD26×
LD XL, ×	DD68	DD69	DD6A	DD6B	DD6F	DD6C	DD6D			DD2E×
LD YH, ×	FD60	FD61	FD62	FD63	FD67			FD64	FD65	FD26×
LD YL, ×	FD68	FD69	FD6A	FD6B	FD6F			FD6C	FD6D	FD2E×

8ビット演算命令

	XH	XL	YH	YL
ADD A, ×	DD84	DD85	FD84	FD85
ADC A, ×	DD8C	DD8D	FD8C	FD8D
SUB ×	DD94	DD95	FD94	FD95
SBC A, ×	DD9C	DD9D	FD9C	FD9D
AND ×	DDA4	DDA5	FDA4	FDA5
XOR ×	DDAC	DDAD	FDAC	FDAD
OR ×	DDB4	DDB5	FDB4	FDB5
CP ×	DDBC	DDBD	FDBC	FDBD
INC ×	DD24	DD2C	FD24	FD2C
DEC ×	DD25	DD2D	FD25	FD2D

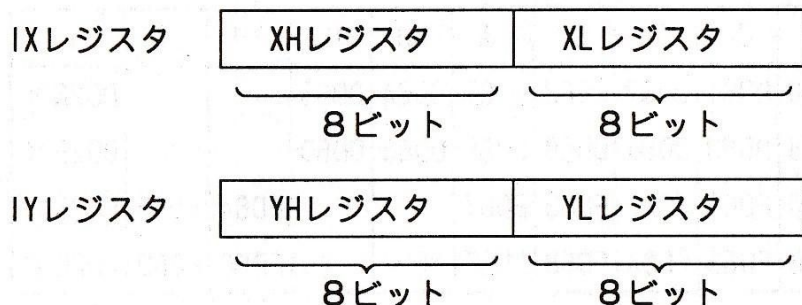
※レジスタ名、記号
XH ････ IX の上位8ビット
XL ････ IX の下位8ビット
YH ････ IY の上位8ビット
YL ････ IY の下位8ビット
× ････ レジスタ、データ

入力命令

IN (C)	ED70

8ビット転送／演算命令

まず、8ビット転送命令と8ビット演算命令の追加された部分というのは、**IX**、**IY**という16ビットレジスタを8ビットごとに分けて使うことができるようになった点です。

XHは**IX**レジスタの上位8ビットで、**XL**は**IX**レジスタの下位8ビットです。同じように**YH**は**IY**レジスタの上位8ビットで、**YL**は**IY**レジスタの下位8ビットです。

入力命令

IN　r,(C) という命令は今までもありました。この命令は **C** レジスタで示すポートから１バイトのデータを入力し、ｒレジスタにセットするという命令でした。**R800**では“**IN　(C)**”の命令が追加されたのですが、これはデータを入力するところまでは“**IN　r,(C)**”と同じなのですが、入力されたデータがどこのレジスタにもセットされない点に注目してください。ただし、入力されたデータによって **S** フラグと **Z** フラグが変化します。

かけ算命令

Z−80ではかけ算ができませんでしたが、**R800**ではかけ算が可能になりました。このかけ算の命令を行うのが **MUL** 命令です。**Z−80**ではかけ算を行うためにそれなりのプログラムを作成しなければいけなかったのですが、その手間がなくなりました。

かけ算は８ビットのかけ算と16ビットのかけ算の２種類があります。８ビットのかけ算の結果は16ビットに、16ビットのかけ算は32ビットになります。いずれのかけ算も符号なしで計算されます。計算された結果によって**Z**フラグとキャリーフラグが変化します。

８ビットかけ算

かけ算は **A** レジスタと **B**、**C**、**D**、**E** レジスタとの間で行い、結果を **HL** レジスタにセットします。

命令の種類	動　作
MUL　B	A × B → HL
MUL　C	A × C → HL
MUL　D	A × D → HL
MUL　E	A × E → HL

16ビットのかけ算

16ビットのかけ算は**HL**レジスタと**BC**または**SP**レジスタとの間で行われ、結果は**DE**、**HL**レジスタに連続してセットされます。

命令の種類	動　作
MUL　BC MUL　SP	HL × BC → DEHL HL × SP → DEHL

＜実習17＞

８ビットのかけ算を実際にプログラムして結果を確かめてみましょう。かけ算するデータは**DATA1**番地のデータと**DATA2**番地のデータで、結果は**ANS**、**ANS+1**番地に格納することにします。

```
100              ORG     8000H
110              LD      A,(DATA1)
120              LD      B,A
130              LD      A,(DATA2)
140              MUL     B
150              LD      (ANS),HL
160              JP      103
170 ;
200     DATA1:   DEFB    12H
210     DATA2:   DEFB    34H
220     ANS:     DEFW    2
230              END
```

実行結果、つまり**ANS**、**ANS**+1番地の内容が“03A8”になっていればプログラムが正しく実行されたことになります。

12H → 18		16進数の12Hは10進数で18。
34H → 52		16進数の34Hは10進数で52。
18×52 ＝ 936		18×52は936
936 → 3A8H		10進数の936は16進数で3A8Hになります。

第7章

疑似命令

この章ではアセンブラプログラムを作成するためによく使われる疑似命令の解説を行っています。

7−1 疑似命令の種類

ここでは *Simple ASM* で使うことのことのできる疑似命令を説明しましょう。

まず、疑似命令とは何かということを説明しましょう。**LD** 命令や **JP** 命令にはその命令に対応したマシン語がありますが、しかし疑似命令には対応するマシン語はありません。疑似命令は **Z**−80自身が持っている命令ではないからです。疑似命令はアセンブラに対して様々な情報を与える命令のことを言います。

疑似命令は以下の種類があります。

ORG
EQU
DEFB(DB)
DEFW(DW)
DEFS(DS)
DEFM(DB)
END

7−2 疑似命令の解説

ORG(Orgin) 命令

この命令はプログラムをどの番地から作成するのかを指定するものです。ひとつのプログラムの中にいくつもの **ORG** 命令を使用しても構いません。

次のプログラムはメインプログラムの部分、データの部分、そしてサブルーチンの部分にわけて、それぞれ **ORG** 命令を入れてみた例です。

メインプログラムは8000H番地から、データ部分は8100H番地から、そして、サブプログラムは9000H番地からマシン語が作成されるようになっています。

```
100         ORG   8000H
110         LD    HL,MSG
120 LOOP:   LD    A,(HL)
130         OR    A
140         JP    Z,EXIT
150         CALL  CHPUT
160         INC   HL
170         JR    LOOP
180 EXIT:   JP    103H
200 ;
210         ORG   8100H
220 MSG:    DEFM  'Hello!!'
230         DEFB   0
300 ;
310         ORG   9000H
320 CHPUT:  LD    E,A
330         LD    C,02H
340         PUSH  AF
350         PUSH  HL
360         CALL  0005H
370         POP   HL
380         POP   AF
390         RET
400         END
```

Simple ASM では、 **ORG** 命令を省略してプログラムを書くと100H番地が開始番地になります。

EQU(Equate) 命令

この命令はラベルに値を割り付ける命令です。

例えば、“CORAL:　EQU　0005H”という１行があれば、**CORAL** に“0005H”が割り当てられ、その後**CORAL**というラベル名を使うことによって

“0005H” という値を使うことができます。

次のプログラムは **ORG** 命令の説明のところで利用したプログラムですが、**EQU** 命令を使っていくつかのラベルに値を定義してみました。次のようにソースプログラムを変更してもオブジェクトプログラムは全く変化はありません。

```
10  MAIN:   EQU  8000H
20  DATA:   EQU  8100H
30  SUBP:   EQU  9000H
40  FUNC:   EQU  0005H
50  SIMPLE:EQU  0103H
100         ORG   MAIN
110         LD    HL,MSG
120 LOOP:   LD    A,(HL)
130         OR    A
140         JP    Z,EXIT
150         CALL  CHPUT
160         INC   HL
170         JR    LOOP
180 EXIT:   JP    103H
200 ;
210         ORG   DATA
220 MSG:    DEFM  'Hello!!'
230         DEFB   0
300 ;
310         ORG   SUBP
320 CHPUT:  LD    E,A
330         LD    C,02H
340         PUSH  AF
350         PUSH  HL
360         CALL  0005H
370         POP   HL
380         POP   AF
390         RET
400         END
```

このようにプログラムの先頭でラベルに値を定義しておけば、プログラムのメンテナンスが簡単になります。

DEFB(Define Byte) 命令

この命令はオペランド覧の値を８ビットのデータとしてオブジェクトプログ

ラムを作成します。オペランド覧にはいくつもの値を続けて書くことができます。そのときは“，（カンマ）”で区切ります。

10進定数の他、２進定数、16進定数、文字定数を記述できます。

次のプログラムをアセンブルしてどのようなオブジェクトプログラムができるか確認してみましょう。

```
100          ORG      9000H
110  DATA1:DEFB     10
120  DATA2:DEFB     10B
130  DATA3:DEFB     10H
140  DATA4:DEFB     'A'
150  DATA5:DEFB     'M','S','X'
```

DATA1番地は10進数の“10”を、DATA2番地には２進数の“10”を、そしてDATA3番地には16進数の“10”を 記述しています。DATA4番地、DATA5番地は文字定数を記述しています。

DEFW(Define Word) 命令

DEFB命令ではオペランド欄の値を８ビットのデータとして扱いましたが、**DEFW**命令では16ビットデータとして扱います。**DEFB**命令と同じように“，（カンマ）”で複数のオペランド記述することができます。

10進定数の他、２進定数、16進定数、文字定数を記述できます。

＜実習18＞

次のプログラムで作成されたオブジェクトプログラムを予想してください。

```
1000          ORG      9000H
1010          DEFB     'A','B'
1020          DEFW     'A','B'
1030          DEFW     'AB'
1040          END
```

作成されると思われるオブジェクトプログラムを書き込んでください。

9000 □ □ □ □ □ □ □ □ □

結果を予想したら、プログラムをアセンブル（**AO**コマンドを使用）し、Ｄコマンドでメモリの内容を表示させてみます。

```
>D9000,9008
```

＜解説＞

まず、文字コード“**A**”は16進数で41H、“**B**”は42Hになることを念頭に入れてください。

DEFB　'A','B'

この命令は先ほど解説したように８ビットのデータが生成されます。従って、41H、42H順にオブジェクトコードが作成されます。

DEFW　'A','B'

カンマで区切られた２つのオペランドがありますので、これは“DEFW　'A'”と“DEFW　'B'”の命令と同じになります。

この命令は16ビットのデータを生成する疑似命令です。

16ビットとは２バイトですから'A'だけで２バイトのコードが作成され、同じように'B'で２バイトコードが作成されます。従って'A'は0041H、'B'は0042Hになります。**Z**−80のオブジェクトコードは上位８ビットと下位８ビットは逆になる規則がありますので、'A'は41H、00Hになり、'B'は42H、00Hになります。

DEFW　'AB'

カンマで区切られていませんから、この１行で２バイトのデータが生成されます。やはり上下バイトが反転しますから42H、41Hの順でオブジェクトコードが作成されます。

<結果>

結果をDコマンドを使って確認しましょう。

```
>D9000,9007
9000 41,42,41,00,42,00,42,41
```

DEFS(Define Storage)命令

オペランドで指定したバイト数分のメモリを確保します。この命令はもう何回も使ったので理解されていると思いますが、もう一度復習しましょう。

DEFS　10

この例は、10バイトの分のメモリが確保されます。

SCORE:DEFS　2

この例は、**SCORE**番地に２バイトの領域を確保する命令です。２バイトの領域に確保したデータは**HL**レジスタなどを使って容易に取り出すことができます。

次の例は、**SCORE**番地のデータを＋１するルーチンです。

```
        LD     HL,(SCORE)
        INC    HL
        LD     (SCORE),DHL
               :
 SCORE: DEFS   2
```

各種のレジスタをペアレジスタとして使っているときにそのデータの保存用として使うと便利でしょう。

DEFM(Define Memory)命令

この命令はオペランド欄に書かれた文字を文字定数としてメモリにセットします。

DEFM 'Hello'

“**H**”、“**e**”、“**l**”、“**l**”、“**o**”の文字コードが順にメモリ内にセットされます。**DEFW** 命令の時のように上位バイトが逆になることはありません。

END命令

アセンブラに対してプログラムがすべて終わったことを指示します。従ってこの命令以降に書かれたものはすべて無効になります。

次のプログラムは140行が **END** 文のためそれ以降の行はアセンブルされません。

```
100       ORG   8000H
110       LD    A,'A'
120       CALL  PUT
130       JP    103H
140       END
150 ;
200 PUT:  CALL  PUTCHR
210             :
                :
```

第8章
ファンクションコール

本章ではMSX－DOSが持っているファンクションコールという機能について解説しています。

8－1 ファンクションコールとは

Simple ASM Ver2.0 はMSX－DOS上で動作するプログラムです。
MSX－DOS上で動作するというのはMSX－DOSの機能を使っているということです。私たちが作成するプログラムも、このMSX－DOSの機能を活用することができるのです。

例えば、「画面に文字を表示する」、「プリンタに文字を印刷する」といった機能はMSX－DOSの機能として用意されているので、わざわざそのプログラムを記述する必要はないのです。

MSX－DOSの機能には「画面に文字を表示する」、「プリンタに文字を印刷する」などの簡単な機能からファイルをアクセスしたりするような高度な機能まで用意されています。MSX－DOSのこれらの機能を利用すれば、私たちの作成するプログラムの負担が少なくなります。

MSX－DOSの機能を利用する手順を“ファンクションコール”と呼びます。この章ではこのファンクションコールの簡単な使い方を解説しています。

8－2 ファンクションコール手順

ファンクションコールを利用するには決められた手順が必要です。すべてのファンクションコールは同じような手順を踏みます。

手順

具体的には次のような手順になります。

①ファンクションコールで必要なレジスタに必要な値をセットする。
②Ｃレジスタに決められたファンクション番号をセットする。
③５番地を **CALL** する。

例えば、画面に一文字表示するという機能はファンクション番号“02H”です。この機能は **E** レジスタの内容を文字コードとして画面に表示するものです。従ってファンクションコールの手順は次のようになります。

```
LD   E,'?'      ?には適当な文字をセット
LD   C,02H      機能02Hを指定
CALL 0005H      ５番地をコール
```

戻り値

ファンクションコールを実行した結果はファンクションコールの種類によってメモリやレジスタに値がセットされるものがあります。
例えばキーボードから文字を入力するファンクションコールでは、入力された文字コードがＡレジスタにセットされて戻ってきます。

レジスタ内容は保証されない

ファンクションコールはサブルーチンをコールするように **CALL** 命令を使います。従ってファンクションコールが終了した時点では **CALL**　0005H命令の次の命令が実行されるわけですが、このときに注意しなければいけないのは、レジスタの内容です。ファンクションコールを実行する前とファンクションコールを実行したあとでは、レジスタの内容が変わっているということです。もし、ファンクションコール前のレジスタの内容が必要なときには必ず、メモリに保存するか、 **PUSH** 命令使ってスタック領域に退避させておきます。

8－3 各種ファンクションコール

すべてのファンクションコールを解説するには紙面が足りませんので本書では代表的なものだけを解説しますが、とりあえずどんなものがあるかだけ表にしてみました。

番号	機　　能
00	システムリセット
01	コンソール１文字入力
02	コンソール１文字出力
03	補助入力装置１文字入力
04	補助出力装置１文字出力
05	プリンタ１文字出力
06	コンソール直接入出力（入力待ちなし、エコーバックなし、コントロールキャラクタの処理なし。）
07	コンソール直接入力 （エコーバックなし、コントロールキャラクタの処理なし。）
08	コンソール１文字入力（エコーバックなし）
09	コンソール文字列出力
0A	コンソール１行入力
0B	コンソール入力チェック
0C	バージョン番号の取り出し
0D	ディスクリセット
0E	デフォルトドライブの設定
0F	ファイルのオープン
10	ファイルのクローズ
11	ファイルの検索
12	ファイルの検索
13	ファイルの削除

番号	機　　　能
14	シーケンシャル読み出し
15	シーケンシャル書き込み
16	ファイルの作成
17	ファイル名の変更
18	ログインベクトルの獲得
19	デフォルトドライブの獲得
1A	転送アドレスの設定
1B	ディスク情報の獲得
21	ランダム呼び出し
22	ランダム書き込み
23	ファイルサイズの獲得
24	ランダムレコードの設定
26	ランダムブロックの書き込み
27	ランダムブロックの読み出し
28	ゼロ書き込みを伴うランダム書き込み
2A	日付の獲得
2B	日付の設定
2C	時刻の獲得
2D	時刻の設定
2E	ベリファイフラグの設定
2F	論理セクタの読み出し
30	論理セクタの書き込み

8−4 代表的なファンクションコールについて

システムリセット（ファンクションコード“00H”）

コール手順：　Cレジスタ　←　00H
戻り値　　：　なし

このファンクションコールが実行されると **MSX−DOS** がウォームスタートとし、**MSX−DOS** のコマンド待ちの状態になります。

ここの“コール手順”とは、ファンクションコールをする前に行っておかなければいけない処理のことです。

“戻り値”とはファンクションコールを実行した結果、どのレジスタがどんな内容になっているかを表します。ここに書かれていないレジスタの内容は保証されていません。

コンソール入力（ファンクションコード“01H”）

コール手順：　Cレジスタ　←　01H
戻り値　　：　Aレジスタ　←　コンソールから入力された文字。

コンソールとは入力のためのキーボードと出力のための画面、プリンタのことです。　このコンソール入力はキーボードから１文字を **A** レジスタにセットするものです。入力された文字は画面にも表示されます。

CTRL + C このキーが押されるとプログラムを中断し、**MSX−DOS** に戻ります。

CTRL + P このキーが押されると、この後のコンソール出力はプリンタにもエコーバックします。

CTRL + N プリンタの出力を終了します。

用語解説
エコーバック：入力した文字がそのまま画面に表示されること。

コンソール出力（ファンクションコード“02H”）

コール手順：	Cレジスタ	←	02H	
	Eレジスタ	←	出力する文字コード	
戻り値　：	なし。			

Eレジスタにセットされたデータを文字コードとしてコンソールに出力します。

コンソールに出力する際にキーボードのチェックも行われ、CTRL+C、CTRL+P、CTRL+Nの処理が有効になります。

CTRL+Sのキーが押されると次に何かキーが押されるまでプログラムは一時中断します。

プリンタ出力（ファンクションコード“05H”）

コール手順：	Cレジスタ	←	05H
	Eレジスタ	←	出力する文字コード
戻り値　：	なし。		

プリンタに文字を印刷するファンクションコールです。

直接コンソール入出力（ファンクションコード“06H”）

```
コール手順：  Cレジスタ  ←  06H
              Eレジスタ  ←  FFH       コンソールからの入力
              Eレジスタ  ←  FFH以外   出力する文字コード
戻り値    ：  入力時。  入力があるとき  Aレジスタ ← 文字コード
                        入力がないとき  Aレジスタ ← “00H”
              出力時。  なし。
```

このファンクションコールはコールするときのEレジスタの内容によって“コンソール入力”として機能したり、“コンソール出力”になったりします。

入力として機能させるには、EレジスタにFFHをセットします。出力として機能させるためにはFFH以外の文字コードEレジスタにセットします。

ただし、コンソール入力機能として働かせる場合は、今までに説明したコンソール入力機能と違って、文字が入力されなくてもファンクションコールは終

了します。またエコーバックの機能は働きません。

直接コンソール入力（ファンクションコード“07H”）

コール手順：　Cレジスタ ← 07H
戻り値　　：　Aレジスタ ← 文字コード

コンソールから１文字入力するファンクションコールです。
エコーバックとコントロールキャラクタの処理は行いません。

コンソール入力（ファンクションコード“08H”）

コール手順：　Cレジスタ ← 08H
戻り値　　：　Aレジスタ ← 文字コード

同じようなファンクションコールが続きますが、１文字入力はこれが最後です。

このファンクションコールはコンソールから１文字入力するものですが、エコーバックは行われませんがコントロールキャラクタ（**CTRL+C**、**CTRL+P**、**CTRL+N**）の機能は働きます。

＜実習19＞

キーボードから入力され文字ががもし英小文字ならば英大文字にして表示するプログラムをファンクションコール“07”と“02”を使って作成してみましょう。

入力された文字を加工してから表示することになるのですから、エコーバックがされては困ります。従って文字の入力にはファンクションコール“07”を使いました。

リターンキーが押された時点で終了することにします。

プログラムの流れはおよそ次のようになります。

①１文字入力する。

②リターンキーかチェック。リターンキーならば終了。

③小文字か？

④小文字ならば大文字にする。

⑤文字の表示。

⑥ふたたび①から実行。

この中で小文字かどうかをチェックするには入力された文字を“'a' よりも小さいか？”ということと“'z' よりも大きいか？”ということで確認します。両方の条件を満たさなければ、'a' ～ 'z' の範囲にあるということになります。

③－１　入力された文字は 'a' よりも小さいか？　小さければ⑤へ

③－２　入力された文字は 'z' よりも大きいか？　大きければ⑤へ

“入力された文字は 'a' よりも小さいか？”というのは“ **CP 'a'** ”を実行します。ここでキャリーフラグがたてば 'a' よりも小さいということになります。

“入力された文字は 'z' よりも大きいか？”というのは“ **CP 'z'+1**”を実行します。その結果キャリーフラグがたたなければ 'z' よりも大きいということになります。

では、次に英小文字を英大文字にする方法を考えてみましょう。
小文字の 'a' の文字コードは61Hで 'z' の文字コードは7AHです。
大文字の 'A' の文字コードは41Hで 'Z' の文字コードは5AHです。つまり英小文字だった場合、文字コードから単純に20Hを引いたものが英大文字の文字コードになります。

```
100 FUNC:       EQU    0005H      }
110 ONEIN:      EQU    07H        } EQUで値を定義しました。
120 ONEOUT:     EQU    02H        }
130 ;
140             ORG    8000H      ― プログラム開始番地
150  LOOP:      LD     C,ONEIN    } １文字入力の
160             CALL   FUNC       } ファンクションコール
170             CP     0DH        } 改行コード（'0D'）ならば
180             JP     Z,103H     } 処理を終了。
```

```
190              CP      'a'         }  'a'より小さければ大文字に
200              JR      C,DISP      }  しない。
210              CP      'z'+1       }  'z'より大きければ大文字に
220              JR      NC,DISP     }  しない。
230              SUB     20H         ―  大文字に変換
240  DISP:       LD      E,A         }
250              LD      C,ONEOUT    }  １文字出力。
260              CALL    FUNC        }
270              JR      LOOP        ―  次の文字を入力。
280              END
```

文字列出力（ファンクションコード“09H”）

コール手順：	Cレジスタ	← 09H
	DEレジスタ	← コンソールに出力する文字列の先頭番地。
戻り値　：	なし。	

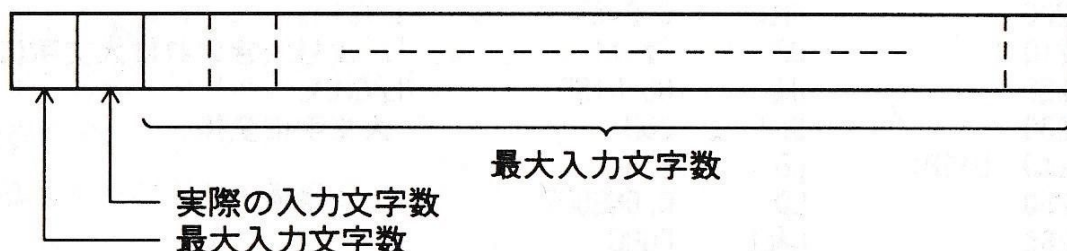
連続した文字列を表示するファンクションコールです。**DE**レジスタに表示させたい文字列の先頭番地をセットしてコールします。

文字列の最後は24H（**'$'**）にしておきます。

文字列入力（ファンクションコード“0AH”）

コール手順：	Cレジスタ	← 0AH
	DEレジスタ	← 行バッファの先頭アドレス。
	(DE)	← 最大入力文字数（１～255まで）
戻り値　：	(DE+1)	← 入力文字数
	(DE+2)以降	← 入力された文字列

このファンクションコールはキーボードからの文字列入力です。文字列を入力しリターンキーを押されるとファンクションコール終了します。

まず、ファンクションコールを行う前にキーボードから入力された文字をメモリ上のどこの領域に保存するのかを決めておきます。この領域のことを“行バッファ”と呼びます。

行バッファの大きさは最大文字数の＋２バイト分用意します。行バッファの先頭は何バイト入力するかを表す“最大文字入力数”です。これはファンクションコールする前にセットしておかなければいけません。

ファンクションコールを終えると、行バッファの２バイト目に実際に入力された文字数がセットされます。そして、３バイト目からが実際に入力された文字列がセットされます。入力される文字列はリターンキーの直前までになります。最大入力文字数以上の入力はできません。

コントロールキャラクタはチェックされます。

＜実習20＞

先ほどの英小文字が入力されたら英大文字に変換するプログラムをもとに、１行単位で変換するプログラムを作成してみましょう。

もちろん、文字列入力と文字列出力を使います。

```
100 FUNC:       EQU    0005H     ;ﾌｧﾝｸｼｮﾝｺｰﾙ ﾊﾞﾝﾁ ﾉ ﾃｲｷﾞ
110 STRIN:      EQU    0AH       ;ﾓｼﾞﾚﾂﾆｭｳﾘｮｸ ﾌｧﾝｸｼｮﾝ
120 STROUT:     EQU    09H       ;ﾓｼﾞﾚﾂｼｭﾂﾘｮｸ ﾌｧﾝｸｼｮﾝ
130 ;
140             ORG    8000H     ;ﾌﾟﾛｸﾞﾗﾑ ﾉ ｶｲｼﾊﾞﾝﾁ
150  LOOP:      LD     DE,BUFF   ;ｷﾞｮｳﾊﾞｯﾌｧ ﾉ ｾﾝﾄｳﾊﾞﾝﾁ ｦ DE ﾆ
160             LD     C,STRIN   ;ﾓｼﾞﾚﾂﾆｭｳﾘｮｸ
170             CALL   FUNC      ;
180             CALL   CRLF      ;ｶｲｷﾞｮｳｼｮﾘ
190             LD     HL,BUFF+1 ;ﾆｭｳﾘｮｸ ﾓｼﾞｽｳﾊ?
200             LD     A,(HL)    ;0?
210             OR     A         ;
220             JP     Z,103H    ;0ﾅﾗﾊﾞ ｼｭｳﾘｮｳ
230 ;
240 CHECK:      LD     HL,BUFF+1 ;ﾆｭｳﾘｮｸﾓｼﾞｽｳｦ Bﾚｼﾞｽﾀ ﾆ
```

```
250              LD      B,(HL)      ;
260 CHECK1:      INC     HL          ;ﾆｭｳﾘｮｸｻﾚﾀ ﾓｼﾞｦ ｼﾞｭﾝﾆ Aﾚｼﾞｽﾀ ﾆ
270              LD      A,(HL)      ;ｺﾓｼﾞ ﾉ ﾁｪｯｸ
280              CP      'a'         ;'a'ﾖﾘﾁｲｻｲｶ?
290              JR      C,NON       ;
300              CP      'z'+1       ;'z'ﾖﾘｵｵｷｲｶ?
310              JR      NC,NON      ;
320              SUB     20H         ;ｵｵﾓｼﾞﾆ
330              LD      (HL),A      ;ｷﾞｮｳﾊﾞｯﾌｧ ﾆ ｶｸﾉｳ
340 NON:         DJNZ    CHECK1      ;ﾓｼﾞｽｳﾌﾞﾝﾀﾞｹ  ｸﾘｶｴｽ
350 ;
360 DISP:        LD      HL,BUFF+1   ;HLﾚｼﾞｽﾀ ﾆ ﾆｭｳﾘｮｸﾓｼﾞｽｳ ﾉ ﾊﾞﾝﾁ
370              LD      C,(HL)      ;ﾆｭｳﾘｮｸﾓｼﾞｽｳ ｦ Cﾚｼﾞｽﾀ ﾆ
380              LD      B,0         ;Bﾚｼﾞｽﾀｦ 0 ﾆ
390              INC     HL          ;HLﾚｼﾞｽﾀ ﾆ ﾆｭｳﾘｮｸﾓｼﾞ ﾉ ｾﾝﾄｳｲﾁ
400              ADD     HL,BC       ;HL+BC=HL
410              LD      (HL),'$'    ;ﾆｭｳﾘｮｸｻﾚﾀ ﾓｼﾞﾉ ｻｲｺﾞﾆ '$'ｦｾｯﾄ
420              LD      DE,BUFF+2   ;ﾓｼﾞﾚﾂﾉ ｻｲｼｮﾉｲﾁｦ DEﾚｼﾞｽﾀﾆ ｾｯﾄ
430              LD      C,STROUT    ;ﾓｼﾞﾚﾂｼｭﾂﾘｮｸ
440              CALL    FUNC        ;
450              CALL    CRLF        ;ｶｲｷﾞｮｳｼｮﾘ
460              JP      LOOP        ;
470 ;
480 CRLF:        LD      DE,CRLFDT   ;ｶｲｷﾞｮｳﾉﾀﾒﾉ ﾓｼﾞﾚﾂｦ DEﾚｼﾞｽﾀﾆｾｯﾄ
490              LD      C,STROUT    ;ﾓｼﾞﾚﾂｼｭﾂﾘｮｸ
500              CALL    FUNC        ;
510              RET                 ;
520 ;                                ;
530 CRLFDT:      DEFB    0DH,0AH,'$';CR/LFﾉﾃﾞｰﾀ
540 ;
550              ORG     9000H       ;ｷﾞｮｳﾊﾞｯﾌｧ ﾉ ｾﾝﾄｳﾊﾞﾝﾁ
500 BUFF:        DEFB    255         ;ｻｲﾀﾞｲﾆｭｳﾘｮｸﾓｼﾞ ｦ 255ﾓｼﾞﾆ
510              DEFS    1           ;ﾆｭｳﾘｮｸ ﾓｼﾞｽｳ
520              DEFS    255         ;ﾓｼﾞﾆｭｳﾘｮｸﾃﾞｰﾀ
530              DEFS    1           ;255ﾓｼﾞﾆｭｳﾘｮｸｻﾚﾀﾄｷﾆ '$' ｶﾞｾｯﾄ
540  ;
550              END
```

このプログラムにはコメントを付加しました。コメントを書くことによって何のためにどのような命令を書いたかがよくわかります。

長いプログラムになればなるほどコメントが役に立ちます。
プログラムの解説は重要な部分のみ行っていますが、それ以外の箇所はコメン

トを参考にしてください。

最初は行バッファについて説明します。行バッファは9000H番地から確保しています。この9000H番地に“**BUFF**”というラベルをつけています。**BUFF**番地には、最大入力文字数をセットしています。このプログラムは255文字にしています。**DEFB**命令になっている点に注意してください。

BUFF+1番地とは**BUFF**番地の次の番地です。つまり9001H番地です。ここに入力された文字数が入ります。**DEFS**命令で1バイトの領域が確保されています。

BUFF+2番地からの255バイトが入力された文字が入る領域です。**DEFS**命令で255バイトの領域が確保されています。

文字列入力のファンクションコールすると行バッファに実際に入力された文字数と入力されたデータがセットされます。もしリターンキーだけが押されると、文字数が“0”になりますから、ここの値をチェックし“0”ならば、プログラムを終了するようにします。

では、文字列の表示を考えてみましょう。文字列の出力は行バッファの入力された文字が記憶されている領域を表示すればよいことになります。文字が入力された領域は**BUFF+**2番地、つまり9002H番地です。文字列の最後は、9002H番地から**BUFF+**1の値（実際に入力文字数）までです。番地でいうと9002H番地＋**BUFF+**1の内容になります。この計算を**HL**レジスタと**BC**レジスタを使って行っています。そして文字列の最後に**'$'**を付加する作業も同時に行っています。（360行～410行まで。）

文字列が入力されたら改行、変換された文字列を表示したら改行をおこなっています。この処理はサブルーチン化され、**CRLF**というルーチンになっています。

以上がファンクションコールを利用した例です。他にもファンクションコールはありますので機会をみつけてチャレンジしてみましょう。

第9章
BIOS

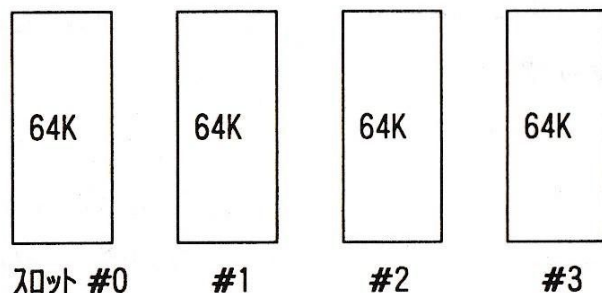
MSX 本体内の **ROM** の中にもいくつかの基本的なプログラムが入っています。これを **BIOS** と呼びますが、本章ではこの **BIOS** についての解説を行ってます。

9―1 スロットについて

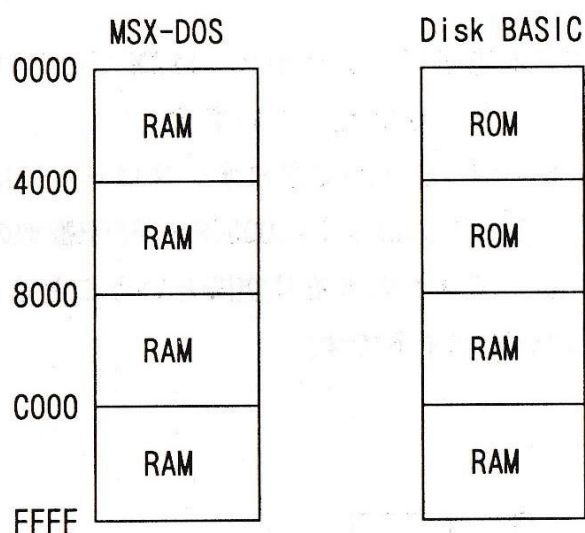
MSX のメモリを **CPU** 側から見ると常に64Kバイトですが、実際にはもっと多くのメモリを内蔵しています。では、 **MSX** ではどのようにメモリを管理しているのでしょう？

基本スロットという言葉を聞いたことがあるでしょうか？　**MSX** ではスロットという考え方で、64**K** バイト以上のメモリを管理しています。

１つの **MSX** は最大で４つのスロットを持つことができます。スロットには#0～#4のスロット番号がふられます。１つのスロットは0000H～FFFFH番地の64**K** バイトのメモリ空間になります。従って４つのメモリ空間ということは４個×64**K** バイトになります。つまり256**K** バイトです。

スロット

また、１つのスロット中でさらに４つのスロットのメモリを管理することもできます。これを拡張スロットと呼んでいます。この拡張スロットも１つのスロットあたり64**K** バイトのメモリ空間になります。

ですから、拡張スロットを使えば1つのスロットで最大4つの拡張スロットが管理できることになります。全体では「64Kバイト　×　4　×　4　＝　1Mバイト」のメモリを使うことができるのです。

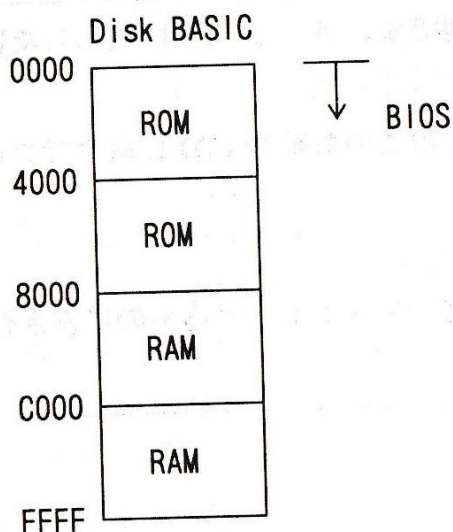
スロットについての解説はここではこの辺でやめておきましょう。そこで、ちょっと確認しておきたいことですが、**MSX−DOS**を使っているときは64Kバイトのメモリ空間がすべて**RAM**になっています。しかし、**Disk BASIC**を使っているときは32Kバイトが**ROM**で**RAM**は残りの32Kバイトになります。これは使っているスロットが**MSX−DOS**と**Disk BASIC**のときとでは異なるということです。

メモリマップ

BIOS

前章でファンクションコールについて説明しましたが、ここでは本章では**BIOS**について解説します。**BIOS**もファンクションコールと同じようにプログラムの基本となるルーチンがいくつか用意されています。ファンクションコールはフロッピィディスクのアクセスが中心となるものですが、**BIOS**は基本的な入出力から画面制御などの機能を持っています。スプライトなどを使っ

たプログラムなどを作るときにはこの **BIOS** のお世話になることでしょう。

Simple ASM のマニュアルの中に掲載されているサンプルプログラムはこれらの **BIOS** を利用しています。

BIOS は0000H番地から始まる **ROM** の中にあるルーチンです。

Disk BASIC 上で **BIOS** の機能を利用するときには **CALL** 命令を使います。しかし、 **MSX−DOS** 上で **BIOS** の機能を利用するときは **CALL** 命令ではできません。 **CALL** 命令では **RAM** の内容をコールすることになってしまいます。ですから、 *Simple ASM* のマニュアルに掲載されているサンプルプログラムを **MSX−DOS** 版の *Simple ASM* では実行できないのです。

インタースロットコール

MSX−DOS 上で **BIOS** の機能を利用するにはインタースロットコールと呼ばれる機能を使います。

例えば画面に１文字出力するプログラムを **BIOS** の機能を使って作ってみましょう。

まず、 **Disk BASIC** 版の *Simple ASM* では次のようになります。

```
100  CHPUT:   EQU   00A2H
110  ASM:     EQU   9000H
120           ORG   0D000H
130           LD    A,'1'
140           CALL  CHPUT
150           JP    ASM
```

１文字出力する**BIOS**は00A2H番地です。**A**レジスタに表示したいデータをセットしておきます。

Disk BASIC版の*Simple ASM*ではこの番地を**CALL**命令でコールするだけでした。

次に、このプログラムを**MSX−DOS**上で動作するものに書き替えてみましょう。

```
100  CHPUT:   EQU   00A2H
110  ASM:     EQU   0103H
120  CALSLT:  EQU   001CH
130  EXPTBL:  EQU   0FCC1H
140           ORG   08000H
150           LD    A,'1'
160           LD    IY,(EXPTBL-1)  ┐
170           LD    IX,CHPUT       ├ この部分が異なる。
180           CALL  CALSLT         ┘
190           JP    ASM
```

インタースロットコールを利用するには、FCC1H-1番地（つまりFCC0H番地）とFCC1H番地の内容を**IY**レジスタにセットし、利用したい**BIOS**の番地を**IY**レジスタにセットし、001CH番地をコールします。

FCC1H番地を**EXPTBL**と名前をつけ、001CH番地を**CALSLT**と名前をつけておきます。そして、**BIOS**を利用するプログラムの先頭に次の２行を記述するようにします。

```
     CALSLT: EQU   001CH
     EXPTBL: EQU   0FCC1H
```

そして、**BIOS**をコールしたいときには、次の３行を無条件に記述するよ

うにすればよいでしょう。

```
        LD      IY,(EXPTBL-1)
        LD      IX,??????          ←  ??????はBIOSの番地
        CALL    CALSLT
```

9−2 ＢＩＯＳを使ったプログラム

Simple ASM のマニュアルに掲載されたプログラムと同等な機能を実現するプログラムを作成してみましょう。

ここでは「 **Sample Program Library** 」の12ページに掲載された **PSG** サウンドプログラムを **MSX−DOS** 用に作りなおしてみましょう。

```
1000 ;
1010 ; PSG Sample Program
1020 ;
1030 ASM:     EQU  0103H
1040 ;
1050 INIPSG: EQU  0090H
1060 WRTPSG: EQU  0093H
1070 ;
1080 CALSLT: EQU   001CH
1090 EXPTBL: EQU   0FCC1H
1100 ;
1110          ORG   8000H
1120 ;
1130 ; INITIAL PSG
1140          LD     A,7
1150          LD     E,11111110B
1160          LD     IY,(EXPTBL-1)
1170          LD     IX,WRTPSG
1180          CALL   CALSLT
1190 ;
1200          LD     A,8
1210          LD     E,15
1220          LD     IY,(EXPTBL-1)
1230          LD     IX,WRTPSG
1240          CALL   CALSLT
```

```
1250 ;
1260            LD    BC,0
1270 ;
1280 ; LOOP
1290 LOOP:      XOR   A
1300            LD    E,B
1310            LD    IY,(EXPTBL-1)
1320            LD    IX,WRTPSG
1330            CALL  CALSLT
1340 ;
1350            DEC   BC
1360            LD    A,B
1370            OR    C
1380            JR    NZ,LOOP
1390 ;
1400            JP    ASM
1410 ;
1420            END
```

どこをどう直すかはもう理解できたことだと思いますが念のために...
マニュアルの“ **CALL WRTPSG** ”の箇所が次のようになっています。

```
        LD     IY,(EXPTBL-1)
        LD     IX,WRTPSG
        CALL   CALSLT
```

もちろん、プログラムの先頭番地や、プログラムを終了したときの戻り番地も変わっていますので注意してください。

Appendix

主要ＢＩＯＳ一覧表
サンプルプログラム各種
コーディングシート

エントリー名	番地	内容	引き渡すレジスタ	受け取るレジスタ	変化するレジスタ
DCOMPR	0020H	HLレジスタとDEレジスタの比較	HL,DE	Cfg,Zfg	AF
INIFNK	003EH	ファンクションキーを初期化する	なし	なし	すべて
DISSCR	0041H	画面表示を停止する	なし	なし	AF,BC
ENASCR	0044H	画面表示停止の解除をする	なし	なし	AF,BC
WRTVDP	0047H	VDPレジスタの書き込みをする Bにデータ,CにVDPレジスタ番号を入れてコールする	BC	なし	AF,BC
RDVRM	004AH	HLで指定したVRAMの内容を読み出してAに入れる	HL	A	AF
WRTVRM	004DH	HLで指定したVRAMへAの内容を書き込む	A,HL	なし	AF
SETRD	0050H	VDPを読み出し状態に設定する	HL	なし	AF
SETWRT	0053H	VDPを書き込み状態にする	HL	なし	AF
FILVRM	0056H	HLで指定したVRAMアドレスから,BCの長さだけAの内容でうめる	A,BC,HL	なし	AF,BC
LDIRMV	0059H	VRAMからのブロック転送を行う HLでVRAMのアドレス DEで目的アドレス BCで長さを指定する	BC,DE,HL	なし	すべて
LDIRVM	005CH	VRAMへのブロック転送を行う HLでソースアドレス DEでVRAMアドレス BCで長さを指定する	BC,DE,HL	なし	すべて
CHGMOD	005FH	SCRMOD (FCAFH) にスクリーンモードの番号を入れてコールする 0でテキストモード,(32×24) 1で(40×24)テキストモード 2でハイレゾリューションモード 3でマルチカラーモードに設定される	WORKによる	なし	すべて
CHGCLR	0062H	カラーモードの変更 FORCLR(F3E9)でフォアグランドカラー BAKCLR (F3EA) でバックグランドカラー BGRCLR (F3EB) でボーダーカラーを設定する	WORKによる	なし	すべて

エントリー名	番地	内容	引き渡すレジスタ	受け取るレジスタ	変化するレジスタ
CLRSPR	0069H	スプライトを初期化する	WORKによる	なし	すべて
INITXT	006CH	テキストモード(40×24)画面の初期化 VDP，ワークエリアの設定	TXTNAM TXTCGP	なし	すべて
INIT32	006FH	テキストモード(32×40)画面の初期化 VDP，ワークエリアの設定	T32NAM T32CGP T32COL T32ATR T32PAT	なし	すべて
INDGRP	0072H	ハイレゾリューションモードの画面初期化 VDP，ワークエリアの設定	GRPNAM GRPCGP GRPCOL GRPATR GRPPAT	なし	すべて
INIMLT	0075H	マルチカラーローレゾリューションモード画面の初期化 VDP，ワークエリアの設定	MLTNAM MLTCGP MLTCOL MLTATR MLTPAT	なし	すべて
SETTXT	0078H	VDPをテキストモード(40×24)にセットする	TXTNAM TXTCGP	なし	すべて
SETT32	007BH	VDPをテキストモード(32×24)にセットする	T32NAM T32CGP T32COL T32ATR T32PAT	なし	すべて
SETGRP	007EH	VDPをハイレゾリューションモードにする	GRPNAM GRPCGP GRPCOL GRPATR GRPPAT	なし	すべて
SETMLT	0081H	VDPをマルチカラーローレゾリューションモードにする	MLTNAM MLTCGP MLTCOL MLTATR MLTPAT	なし	すべて
CALPAT	0084H	スプライトパターンテーブルのアドレスを返す．Aにスプライト IDを入れておく	A	HL	AF，DF，HL
CALATR	0087H	スプライト属性テーブルのアドレス計算 Aレジスタにスプライト IDをセットする	A	HL	AF，DE，HL

エントリー名	番地	内　　容	引き渡すレジスタ	受け取るレジスタ	変化するレジスタ
GSPSIZ	008AH	現在のスプライトサイズをAに入れて返す. 16×16の時にはCfgをセットして返す	なし	A,Cfg	AF
GRPPRT	008DH	GRPACX(FCB7), GRPACY(FCB9)で位置, Aにキャラクタコードを入れてコールするとグラフィックスクリーンに文字が出力される	A及び WORK	なし	なし
GICINI	0090H	PSGの初期化	なし	なし	すべて
WRTPSG	0093H	Aにレジスタ番号, Eにデータを入れてコールすると, PSGレジスタにデータを書き込む	A,E	なし	なし
RDPSG	0096H	Aにレジスタ番号を入れてコールすると, PSGレジスタの内容を読み出す	A	A	A
CHSNS	009CH	キーボードバッファに文字があれば, Zfgをリセットする	なし	Zfg	AF
CHGET	009FH	入力待ちキーボード1文字入力	なし	A	AF
CHPUT	00A2H	画面1文字出力. CSRX, CSRYでロケーション指定. 実行後次の位置 CSRX (F3DC), CSRY (F3DD)	A及び WORK	なし	なし
LPTOUT	00A5H	プリンタ1文字出力. Cfgがセットされて帰るとアボートされたことになる	A	Cfg	F
LPTSTT	00A8H	プリンタステータスチェック Ready ⇨ Zfgリセット, BUSY ⇨ Zfgセット	なし	Zfg	AF
CNVCHR	00ABH	グラフィックヘッダバイトとコンバートグラフィックコードのチェックを行う グラフィックヘッダバイト =Cfgのリセット コンバートグラフィックコード =C,Zfgのセット コンバートされていない =Cfgセット =Zfgリセット	A	Cfg,Zag	AF
PINLIN	00AEH	画面の左端から1行入力を行う. (CRキーまたはSTOPキーが入力されるまでの入力) HLにはバッファの先頭アドレス－1が入り, STOPキーが押されたときはCfgをセット	なし	HL,Cfg	すべて

エントリー名	番地	内容	引き渡すレジスタ	受け取るレジスタ	変化するレジスタ
INLIN	00B1H	1行入力．HLにはバッファの先頭アドレス－1が入る．STOPキーが押された場合はCfgをセットする	なし	HL,Cfg	すべて
QINLIN	00B4H	画面に3FH，20H（？␣）を出力してINLINに移る	なし	HL,Cfg	すべて
BREAKX	00B7H	CTRL＋STOPキーのチェック 押されている＝Cfgをセット 押されていない＝Cfgのリセット	なし	HL,Cfg	AF
BEEP	00C0H	ブザーを鳴らす	なし	なし	なし
CLS	00C3H	画面クリア	なし	なし	AF,CB,DE
POSIT	00C6H	カーソルロケーション．HにX，LにYを入れてコール	HL	なし	AF
FNKSB	00C9H	ファンクションキー表示がアクティブのとき表示し，そうでなければなにもしない	FUKFLG	なし	すべて
ERAFNK	00CCH	ファンクションキー表示クリア	なし	なし	すべて
DSPFNK	00CFH	ファンクションキー表示	なし	なし	すべて
GTSTCK	00D5H	ジョイスティックの状態を返す A⇐0でコールするとキーボード A⇐1でコールするとスティック#1 A⇐2でコールするとスティック#2	A	A （0～8）	すべて
GTTRIG	00D8H	トリガボタンの状態を返す A⇐0でコールするとスペースキー A⇐1でコールするとスティック#1A A⇐2でコールするとスティック#1B A⇐3でコールするとスティック#2A A⇐4でコールするとスティック#2B 押されている⇒255 押されていない⇒　0	A	A （0，255）	AF
GTPAD	00DBH	タッチパッドの状態を返す．AにタッチパッドID（0～7）を入れるとAに値が返ってくる	A	A （0～255）	すべて
GTPDL	00DEH	パドルの状態を返す AにパドルID（0～12）を入れるとAに値が返ってくる	A	A	すべて
TAPION	00E1H	モータをONし，テープのヘッダを読む 異常終了はCfgをセット	なし	Cfg	すべて
TAPIN	00E4H	CMT 1文字入力　アボートされるとCflgをセット	なし	A,Cfg	すべて
TAPIOF	00E7H	テープからの読み込みを停止	なし	なし	なし

エントリー名	番　地	内　　　容	引き渡すレジスタ	受け取るレジスタ	変化するレジスタ
TAPOON	0 0 E A H	モータを ON テープのヘッダを書く ロングヘッダ　A≠0 ショートヘッダ　A＝0 異常終了は Cfg をセット	A	Cfg	すべて
TAPOUT	0 0 E D H	CMT 1文字出力　アボートされると Cfg をセット	A	Cfg	すべて
TAPOOF	0 0 F 0 H	テープへの書き込みを停止	なし	なし	なし
STMOTR	0 0 F 3 H	モーターの状態をセット A⇐0でコールするとモータストップ A⇐1でコールするとモータスタート A⇐FFでコールすると反転	A	なし	AF
CHGCAP	0 1 3 2 H	A ⇐0でコールすると CAP ランプ消灯 A ⇐0以外でコールすると CAP ランプ点灯	A	なし	AF
CHGSND	0 1 3 5 H	1 bit サウンドポートの制御 A＝0で OFF A≠0で ON	A	なし	AF
SNSMAT	0 1 4 1 H	キーボードマトリクスのローアドレスを A に入れてコールするとその状態を返す	A	A	AF
OUTDLP	0 1 4 D H	TAB をスペースに変換し，MSX 仕様でないプリンタに対してはひらがなやグラフィックキャラクタを変換する 異常終了は “Device I/O error” を出力 あとは LPTOUT と同じ働き	A	なし	F
KILBUF	0 1 5 6 H	キーボードバッファをクリアする	なし	なし	HL

数字

アルファベット

ア～オ

カ～コ

サ～ソ

タ～ト

ナ～ノ

ハ～ホ

マ～モ

ラ～ン

サンプルプログラム

サンプルプログラム１

１～100の合計を求めるプログラム。結果は16進数で **ANS** 番地に。

```
1000 N:     EQU    100
1010 ;
1020        ORG    8000H
1030        LD     HL,0
1040        LD     DE,0
1050        LD     B,N
1060 LOOP:  INC    DE
1070        ADD    HL,DE
1080        DJNZ   LOOP
1090        LD     (ANS),HL
1100 ;
1110        JP     103H
1120 ;
1130        ORG    9000H
1140 ANS:   DEFS   2
1150        END
```

サンプルプログラム２

次のプログラムは１～100までの合計を10進数で計算しています。

結果は**ANS**番地に格納されます。結果は9999までになる値でしたら1000行の値を適当な値に替えてもよいでしょう。

```
1000 N:     EQU   100
1010 ;
1020        ORG   8000H
1030        LD    B,N
1040        LD    HL,0
1050        LD    DE,0
1060 LOOP:  CALL  INCDE
1070        LD    A,L
1080        ADD   A,E
1090        DAA
1100        LD    L,A
1110        LD    A,H
1120        ADC   A,D
1130        DAA
1140        LD    H,A
1150        DJNZ  LOOP
1160        LD    (ANS),HL
1170        JP    103H
1180 ;
1190 INCDE:LD     A,E
1200        INC   A
1210        DAA
1220        LD    E,A
1230        LD    A,0
1240        ADC   A,D
1250        DAA
1260        LD    D,A
1270        RET
1280 ;
1290        ORG   9000H
1300 ANS:   DEFS  2
1310 ;
1320        END
```

サンプルプログラム３

Aレジスタの内容を16進数として画面に表示するサブルーチンです。
サブルーチンは180行の **HEXDIS** 番地からです。120行から140行は呼び出すサンプルです。

```
100 X:      EQU   3EH
110         ORG   8000H
120         LD    A,X
130         CALL  HEXDIS
140         JP    103H
150         CALL  HEXDIS
160         JP    103H
170 ;
180 HEXDIS: LD    B,A
190         RRA
200         RRA
210         RRA
220         RRA
230         AND   0FH
240         CALL  HEX2
250         CALL  DISP
260         LD    A,B
270         AND   0FH
280         CALL  HEX2
290         CALL  DISP
300         RET
310 ;
320 HEX2:   ADD   A,30H
330         CP    3AH
340         JR    C,HEX3
350         ADD   A,7H
360 HEX3:   RET
370 ;
380 DISP:   LD    E,A
390         LD    C,2
400         PUSH  BC
410         CALL  5H
420         POP   BC
430         RET
440 ;
450 PUTSP:  LD    E,' '
460         LD    C,2
470         CALL  5H
480         RET
```

さいごに

Z−80のプログラムの作り方がわかっている方が *Simple ASM* を購入し、そしてプログラム作成に利用しているとばかり考えていましたが、実際にはこれからアセンブラを勉強したい人たちのほうが圧倒的に多いようです。この方たちから見れば *Simple ASM* のマニュアルは *Simple ASM* の使い方しか書かれていないので、 *Simple ASM* を買ったのはよいが全く使えないという状態みたいです。そして、当社に多くの方たちから Z−80の入門書を作って欲しいという意見が寄せられました。本書ではそのような人たちの意見に答える形で出版しました。是非、この本でアセンブラを理解して頂ければ筆者としても幸せです。

しかしながら、本書を理解するだけではゲームプログラムを作成できません。本書は Z−80の命令の使い方を中心に解説した本であって、 MSX の機能を説明したわけではないからです。もし、 MSX で高度なプログラムを作成したいのであれば、次に学ぶのは MSX の仕組み、 VDP や SPG などの機能でしょう。そして BIOS についての理解も深めなければいけません。これらのことはアスキー社から出版されている「MSX テクニカルデータブック」などを参考にしてください。

最後に、「入門書が欲しい」と言って頂いた多くの方に、この場を借りてお礼を申し上げます。そして、出版までに時間がかかったことに対してもお詫びも申し上げます。

そして、末永く *Simple ASM* をご利用いただけるようにお願い申しあげます。

以上

ユーザーサポートについて

本書のお問い合わせは、ユーザーサポート電話、ＦＡＸ、郵便にて承っています。尚、ＦＡＸでご質問事項を送付なさった場合でも必ずお電話下さい。

ユーザーサポート電話　03-3587-4685
ＦＡＸ番号　03-3587-6851
受付時間：月～金曜日
午前10時～正午　午後１時から午後５時まで
（祝祭日、夏期休暇、年末年始および弊社行事日は除く。）

MSX で学ぶ Z-80
アセンブラ入門

1993年9月1日　初版発行
1994年3月11日　第２版発行
発行者　那須勇次
発行所
Coral CORPORATION
株式会社 コーラル　〒107 東京都港区赤坂6-4-17 赤坂コーポ 8F
PHONE:03(3587)1481 SUPPORT:03(3587)4685 FAX:03(3587)6851

定価　2,000円　(送料 310円)